

Санкт-Петербургский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

## **Курсовая работа**

**Дисциплина:** Объектно-ориентированное программирование

**Тема:** Алгоритм Ахо-Корасик

Студент группы 3331506/00401

К. К. Машьянов

Преподаватель

М. С. Ананьевский

«\_\_» \_\_\_\_\_ 2023 г.

Санкт-Петербург

2023 г.

## Оглавление

Введение .....	3
Принцип работы .....	4
Оценка скорости и памяти .....	10
Оценка скорости .....	10
Оценка памяти .....	10
Использование красно-чёрного дерева .....	11
Анализ алгоритма .....	12
Зависимость от размера словаря .....	12
Зависимость от размера строки .....	13
Список литературы .....	14
Приложение 1 .....	15

## Введение

Алгоритм Ахо-Корасик — алгоритм поиска подстроки в строке, реализующий поиск множества подстрок из словаря в заданной строке.

Был разработан Альфредом Ахо и Маргарет Корасик в 1975 году.

Используется в множестве системных утилит и системном программировании, в целом. Один из наиболее известных примеров — утилита *grep* операционной системы *Linux*.

Отличительными особенностями данного алгоритма от других алгоритмов поиска подстроки в строке являются:

- возможность множественного поиска по всем подстрокам одновременно;
- отсутствие «срыва» поиска при несовпадении следующего символа со следующим символом в рассматриваемой подстроке.

В рамках данной курсовой работы будет рассмотрена реализация алгоритма на языке программирования C++ с использованием методов объектно-ориентированного программирования.

## Принцип работы

Принцип работы алгоритма заключается в построении конечного автомата, принимающего на вход строку, в которой нужно осуществить поиск заданных подстрок.

Посимвольно получая исходную строку, состояние автомата меняется, переходя по соответствующим рёбрам. В случае, когда автомат приходит в своё конечное состояние, можно утверждать, что искомая подстрока присутствует в заданной строке.

Для поиска по нескольким строкам одновременно стоит создать дерево поиска — т. е. бор, префиксное дерево. Полученный бор является конечным автоматом, который распознаёт одну строку из  $m$ , но при условии, что начало строки известно.

Необходимо обработать случай, когда подстрока не совпала. Перевод автомата в начальное состояние при неподходящей букве не является допустимым, поскольку это может привести к пропуску подстроки. Например, при поиске подстроки *aabac* попадаетея подстрока *aabaabac* — при считывании 5-го символа автомат потребуетея привести в начальное состояние, что приведёт к пропуску подстроки. В данном случае требуется перевести автомат в состояние *a*, после чего снова обработать пятый символ.

Для обработки несовпадения строк вводятся суффиксные ссылки, нагруженные пустым символом. Это превращает детерминированный автомат в недетерминированный. Таким образом, при разборе строки *aaba*, будут суффиксы *aba*, *ba* и *a*. Суффиксная ссылка — это ссылка на тот узел, который соответствует самому длинному суффиксу, который позволяет автомату корректно обрабатывать следующие символы (не заводит автомат в тупик).

Для корневого узла автомата суффиксная ссылка является петлёй, замкнутой на сам корневой узел.

Остальные суффиксные ссылки создаются по следующему алгоритму:

- Последний распознанный символ — *symbol*
- Осуществляется переход по суффиксной ссылке родителя
- Если оттуда есть ребро, нагруженное символом *symbol*,
  - То суффиксная ссылка указывает на тот узел, куда ведёт это ребро
  - Иначе — проход по суффиксной ссылке ещё раз, пока не будет найдено ребро, нагруженное символом *symbol*, либо не будет встречен корень дерева (тогда суффиксная ссылка указывает на корень).

На данном этапе автомат является недетерминированным. Преобразование автомата в детерминированный приведёт к значительному увеличению вершин, в общем случае. Однако, в данный автомат можно сделать детерминированным, не создавая новых вершин. Тогда алгоритм обхода будет выглядеть следующим образом:

- Считываем следующий символ
- Если существует ребро перехода из текущей вершины в следующую — переходим
- Иначе — переходим по суффиксной ссылке и повторяем процесс
- Если пришли в конечную вершину — подстрока присутствует в тексте
- Если пришли в корень — подстрока отсутствует в тексте

Таким образом, количество вершин не увеличивается. Однако, увеличивается количество переходов по суффиксным ссылкам — увеличивается вычислительная сложность алгоритма. Поскольку переход по суффиксным ссылкам, в конечном итоге, ведёт к переходу в конечное состояние, имеет смысл

создать так называемые конечные ссылки. Проход по конечным ссылкам для текущего символа позволяет определить все совпавшие подстроки.

Алгоритм нахождения конечной ссылки выглядит следующим образом:

- Переходим по суффиксной ссылке
- Если текущая вершина является конечной (в т. ч. корнем), то конечная ссылка ссылается на текущую вершину
- Иначе — переходим по суффиксной ссылке текущего корня
- Повторяем до нахождения конечной вершины

Что примечательно — необходимости вычислять все суффиксные и конечные ссылки на этапе создания префиксного дерева нет, вместо этого можно воспользоваться принципом ленивых вычислений, вычисляя ссылки по мере необходимости.

Для поиска всех вхождений каждого элемента из множества подстрок в заданную строку требуется проверять каждый следующий символ на предмет совпадения подстроки. Для этого требуется перейти в следующую вершину и проверить конечные ссылки из данной вершины на предмет совпадения с заданными подстроками. Алгоритм выглядит следующим образом:

- Считать следующий символ
- Перейти в следующую вершину (либо по соответствующему ребру, либо по суффиксной ссылке, если соответствующего ребра не существует)
- Проверить совпадение конечных ссылок (без изменения фактического состояния автомата):

- Если данная вершина является конечной (но не является корнем), вывести сообщение о совпадении соответствующей подстроки
- Перейти по конечной ссылке в следующую вершину
- Повторять, пока текущая вершина не станет корнем
- Повторять до конца строки

На рисунке 1 представлено префиксное дерево для множества подстрок: 1) *acab*, 2) *accc*, 3) *acac*, 4) *baca*, 5) *abb*, 6) *z*, 7) *ac*. Красными окружностями обозначены конечные вершины, попадание в которые свидетельствует о совпадении искомой подстроки. Номер рядом с вершиной показывает, с какой именно подстрокой произошло совпадение.

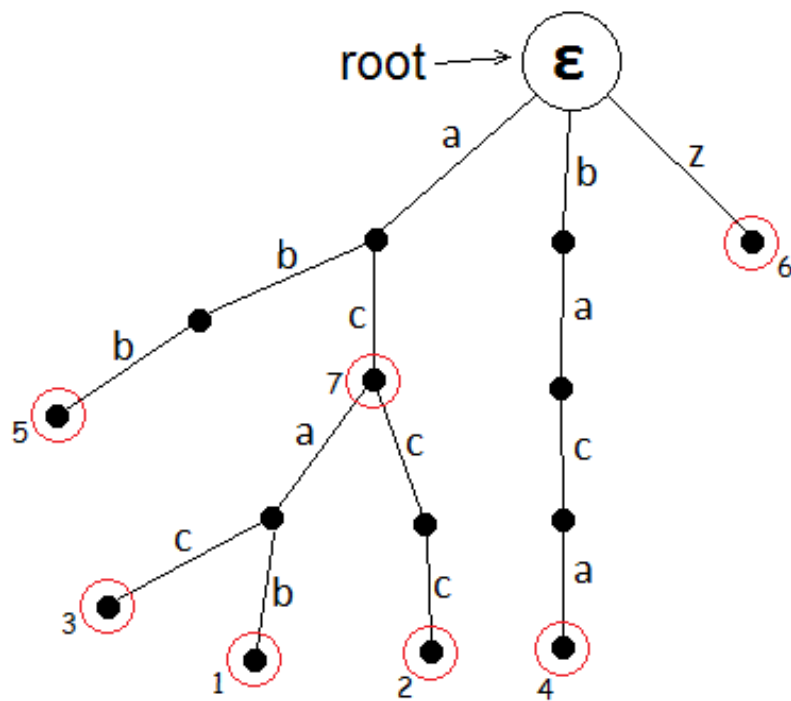


Рисунок 1 — Построение префиксного дерева

На рисунке 2 представлен поиск суффиксной ссылки для текущей вершины *V*, попадание в которую вызвано переходом по ребру *ymb*.

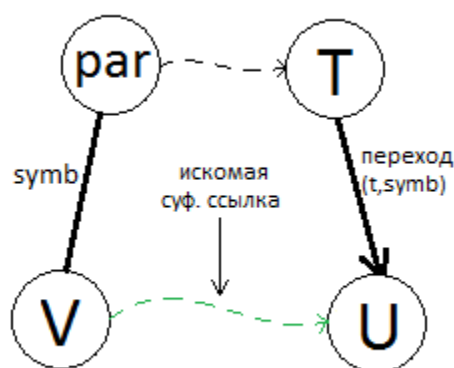


Рисунок 2 — Поиск суффиксной ссылки

На рисунке 3 представлено префиксное дерево из рисунка 1, но с добавленными суффиксными ссылками.

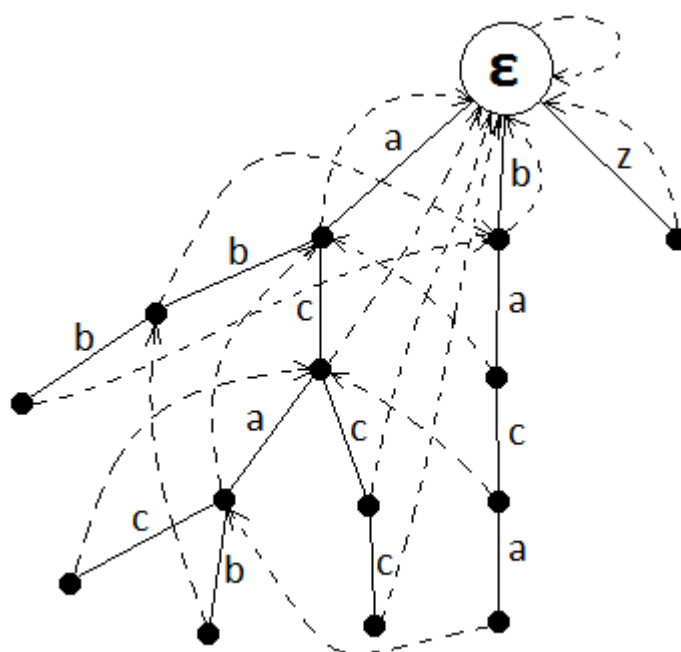


Рисунок 3 — Префиксное дерево с обозначенными суффиксными ссылками

На рисунке 4 представлена визуализация конечных ссылок в сравнении с обычными суффиксными ссылками. Как видно из рисунка, переход по конечной ссылке значительно быстрее перехода по нескольким суффиксным.



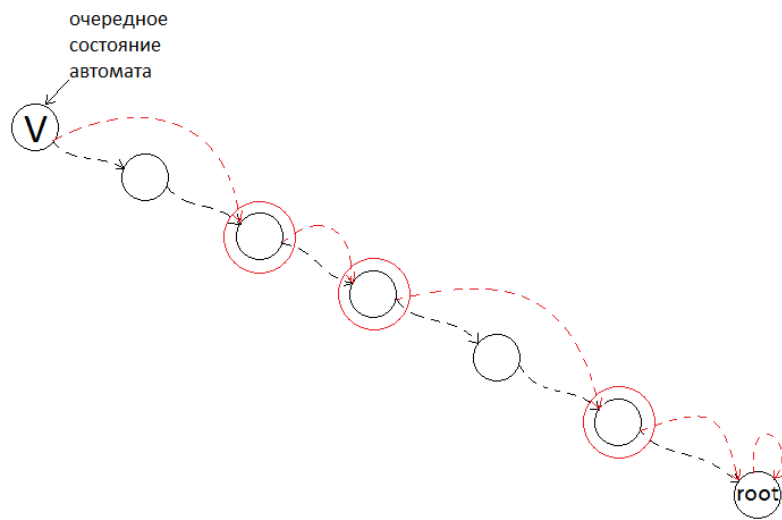


Рисунок 4 — Конечные суффиксные ссылки

На рисунке 5 представлен конечный автомат с изображёнными суффиксными и конечными ссылками. Здесь: серые вершины — промежуточные, белые — конечные, синие стрелки — суффиксные ссылки, зелёные — конечные.

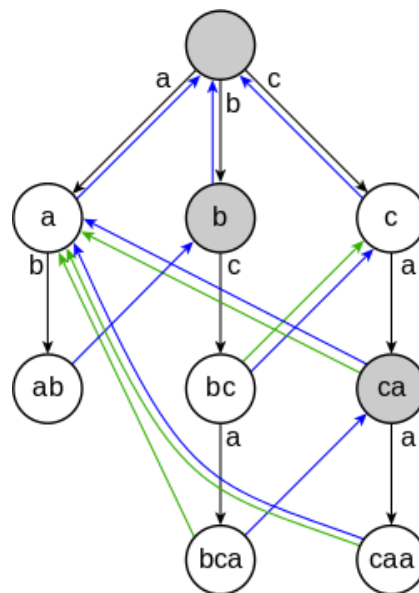


Рисунок 5 — Конечный автомат с показанными ссылками

## Оценка скорости и памяти

### Оценка скорости

Алгоритм целиком проходит по длине строки, равной  $n$ . При этом на каждой итерации цикла выполняется проверка вхождения заданных подстрок в строку на текущей позиции.

Построение префиксного дерева удобно делать на одномерном массиве. Переход по такому массиву

Тогда сложность можно оценить как  $O(n * O(\text{Check}))$ .

Учтём, что функция проверки выполняет переход только по заведомо определённым вершинам (по конечным ссылкам), максимальное количество которых равно количеству подстрок  $m$  в заданном подмножестве.

Переход по таким ссылкам выполняется за линейное время, а максимальное количество переходов равно  $m$ .

Сложность вычисления суффиксных и конечных ссылок пропорциональна общей длине всех подстрок  $l$  в заданном множестве и размеру используемого алфавита  $k$ .

Её можно оценить как  $O(l * k)$ .

Тогда общую сложность можно оценить как  $O(n + m + l * k)$ .

### Оценка памяти

Хранение дерева происходит в массиве размера  $l$ , содержащего все вершины. При этом каждая вершина хранит массив ссылок на другие вершины — рёбра графа — размером, равным  $k$  — длине используемого алфавита.

Тогда используемую память можно оценить как  $O(l*k)$ .

## Использование красно-чёрного дерева

Для уменьшения затрат памяти можно хранить таблицу переходов автомата как красно-чёрное дерево.

В таком случае затраты памяти будут пропорциональны количеству вершин в дереве и могут быть оценены как  $O(l)$

При этом переход по автомату будет происходить уже по дереву, а не по массиву. В таком случае поиск вершины для перехода можно оценить как  $O(n*\log(k))$ .

Тогда общая вычислительная сложность становится  $O((n+l)*\log(k) + m)$ .

Для меньшего расхода памяти и большего диапазона возможных значений символов алфавита был реализован вариант с красно-чёрным деревом, при помощи стандартного класса *map*. При этом количество символов в алфавите резко уменьшится, ведь неиспользуемые символы не будут учтены вовсе. А учитывая логарифмическую зависимость сложности при использовании чёрно-красного дерева, вычислительная сложность при одновременном поиске по большому количеству строк будет меньше, чем при использовании массива.

## Анализ алгоритма

Поскольку алгоритм осуществляет поиск в тексте, для значимой разницы во времени выполнения, требуемой для анализа скорости работы алгоритма, требуются входные данные больших размеров.

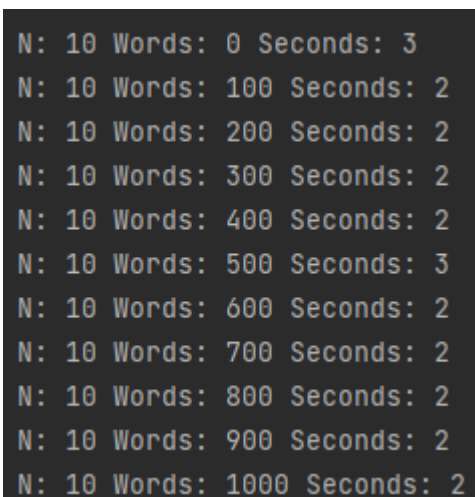
В качестве одной единицы входных данных использован текст книги «Над пропастью во ржи» Джерома Дэвида Сэлинджера. Размер книги составляет 386302 символа. В качестве словаря для поиска подстрок будут использоваться слова из самой книги, идущие в начале книги подряд, для простоты выбора.

При этом функция непосредственно вывода каждого найденного вхождения подстроки выключена, чтобы скорость вывода строки в консоль не учитывалась в скорости работы самого алгоритма.

### Зависимость от размера словаря

Измерим зависимость скорости поиска от размера словаря. При  $N = 10$ , увеличим последовательно размер словаря с шагом 100 от 0 до 1000 слов, где  $N$  — количество повторений книги в заданной строке.

Полученная зависимость:



N	Words	Seconds
10	0	3
10	100	2
10	200	2
10	300	2
10	400	2
10	500	3
10	600	2
10	700	2
10	800	2
10	900	2
10	1000	2

Рисунок 6 — Зависимость времени выполнения от размера словаря

Как видно из рисунка, при любой длине словаря время поиска не изменяется.

### Зависимость от размера строки

Измерим зависимость скорости поиска от длины входной строки. С шагом 10 будем изменять  $N$ , измеряя время поиска подстроки, где  $N$  — размер книги. При этом размер словаря зададим размером в 10 слов.

График полученной зависимости представлен ниже:



Как видно из графика, зависимость можно считать линейной, с учётом погрешности измерений времени встроенными функциями.

## Список литературы

1. Alfred V. Aho, Margaret J. Corasick. Efficient string matching: An aid to bibliographic search // Communications of the ACM. — 1975.
2. Meyer, Bertrand. Incremental string matching — 1985.
3. Интернет ресурс: <https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/> [Дата обращения: 20.04.2023]
4. Интернет ресурс: <https://habr.com/ru/articles/198682/> [Дата обращения: 22.04.2023]
5. Интернет ресурс: [https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм\\_Ахо-Корасик](https://neerc.ifmo.ru/wiki/index.php?title=Алгоритм_Ахо-Корасик) [Дата обращения: 25.04.2023]
6. Интернет ресурс: [https://users.math-cs.spbu.ru/~okhotin/teaching/algorithms\\_2020/okhotin\\_algorithms\\_2020\\_17.pdf](https://users.math-cs.spbu.ru/~okhotin/teaching/algorithms_2020/okhotin_algorithms_2020_17.pdf) [Дата обращения: 26.04.2023]

## Приложение 1 – Код

Код заголовочного файла – «Aho\_Corasick.h»:

```
#ifndef AHO_CORASICK_AHO_CORASICK_H
#define AHO_CORASICK_AHO_CORASICK_H

#include <string>
#include "vector"
#include <map>

using namespace std;

typedef struct{
    map<char, int> next_vertex;    // словарь, связывающий символ ребра с
    номером следующей вершины в дереве
    int pattern_number;          // номер строки-образца в дереве
    int suffix_link;              // суффиксная ссылка
    map<char, int> move;          // словарь, связывающий символ ребра с номером
    следующей вершины в дереве для строки-образца
    int suffix_final_link;        // суффиксная ссылка на финальную вершину
    int parent;                  // номер родительской вершины
    char symbol;                  // символ на ребре, соединяющем текущую вершину с
    родительской
    bool is_final;                // является ли вершина финальной
} tree_vertex;

class AC{
public:
    AC();
    ~AC()= default;
    void add_string(const string &str); // метод добавления строки-образца в
    дерево
    void find(const string &str);      // метод поиска всех строк-образцов в тексте
private:
    vector <tree_vertex> tree;          // вектор вершин дерева
    vector <string> pattern;            // вектор строк-образцов

    static tree_vertex make_vertex(int parent, char symbol); // статический
    метод создания вершины дерева
    int get_suffix_link(int vertex);    // метод получения суффиксной ссылки для
    вершины дерева
    int get_move(int vertex, char character); // метод получения следующей
    вершины в дереве по символу на ребре
    int get_final_suffix_link(int vertex); // метод получения суффиксальной
    ссылки для вершины
    void check(int vertex, int i);      // метод проверки суффиксальных ссылок для
    вершины
};

#endif //AHO_CORASICK_AHO_CORASICK_H
```

Код файла – «Aho\_CorasickK.cpp»:

```
#include <iostream>
#include "Aho_Corasick.h"

AC::AC() {
    this->tree.push_back(make_vertex(0, '$'));
}

void AC::add_string(const string &str) {
    int num = 0;
    for (int i = 0; i < (int) str.length(); i++) {
        char symbol = str[i];
        if(this->tree[num].next_vertex.find(i) == this->tree[num].next_vertex.end()){
            this->tree.push_back(make_vertex(num, (char) symbol));
            this->tree[num].next_vertex[symbol] = (int) this->tree.size() - 1;
        }
        num = this->tree[num].next_vertex[symbol];
    }
    this->tree[num].is_final = true;
    this->pattern.push_back(str);
    this->tree[num].pattern_number = (int) this->pattern.size() - 1;
}

void AC::find(const string &str) {
    int u = 0;
    for (int i = 0; i < (int) str.length(); i++){
        u = get_move(u, (char) (str[i]));
        this->check(u, i + 1);
    }
}

tree_vertex AC::make_vertex(int parent, char symbol) {
    tree_vertex vertex;
    vertex.is_final = false;
    vertex.suffix_link = -1;
    vertex.suffix_final_link = -1;
    vertex.parent = parent;
    vertex.symbol = symbol;
    return vertex;
}

int AC::get_suffix_link(int vertex) {
    if(this->tree[vertex].suffix_link == -1) {
        if (vertex == 0 || this->tree[vertex].parent == 0)
            this->tree[vertex].suffix_link = 0;
        else
            this->tree[vertex].suffix_link = get_move(get_suffix_link(this->tree[vertex].parent), this->tree[vertex].symbol);
    }
    return this->tree[vertex].suffix_link;
}

int AC::get_move(int vertex, char symbol) {
    if(this->tree[vertex].move.find(symbol) == this->tree[vertex].move.end()){
        if(this->tree[vertex].next_vertex.find(symbol) != this->tree[vertex].next_vertex.end()){
            this->tree[vertex].move[symbol] = this->tree[vertex].next_vertex[symbol];
        }
    }
}
```



```

        } else {
            if (vertex == 0)
                this->tree[vertex].move[symbol] = 0;
            else
                this->tree[vertex].move[symbol] =
get_move(get_suffix_link(vertex), (char) symbol);
        }
    }
    return this->tree[vertex].move[symbol];
}

int AC::get_final_suffix_link(int vertex) {
    if(this->tree[vertex].suffix_final_link == -1){
        int u = get_suffix_link(vertex);
        if (u == 0){
            this->tree[vertex].suffix_final_link = 0;
        }
        else{
            if(this->tree[u].isFinal)
                this->tree[vertex].suffix_final_link = u;
            else
                this->tree[vertex].suffix_final_link = get_final_suffix_link(u);
        }
    }
    return this->tree[vertex].suffix_final_link;
}

void AC::check(int vertex, int i) {
    for(int u = vertex; u != 0; u = get_final_suffix_link(u)){
        if(this->tree[u].isFinal){
            std::cout << i - this->pattern[this->tree[u].pattern_number].length() + 1 << " " << this->pattern[this->tree[u].pattern_number] << std::endl;
        }
    }
}
}

```

## Код файла - «main.cpp»:

```
#include <iostream>
#include "Aho_Corasick.h"
#include <ctime>
#include <fstream>

using namespace std;

int main() {

    class AC automat;

    ifstream file;
    char text[100];
    file.open("book.txt");
    string book;
    if(file) {
        while (!file.eof()) {
            file >> text;
            book.append(text);
            book.append(" ");
        }
        file.close();
    }
    // int N = 15;
    int words = 1000;

    file.open("book.txt");

    string source;

    cout << "Size of one book " << book.length() << endl;
    for(auto j = 0; j < 10; j++){
        source.append(book);
    }
    for (auto i = 0; i <= words; i+=100) {

        unsigned long long before = time(nullptr);
        automat.find(source);
        unsigned long long after = time(nullptr);
        cout << "N: " << 10 << " Words: " << i << " Seconds: " << after - before
        << endl;
        for(int k = 0; k < words; k++){
            string temp;
            file >> temp;
            automat.add_string(temp);
        }
    }
    file.close();
    return 0;
}
```