|  |  |  |  |
| --- | --- | --- | --- |
| Санкт-Петербургский политехнический университет Петра Великого  Институт машиностроения, материалов и транспорта  Высшая школа автоматизации и робототехники | | | |
| Курсовая работа  Дисциплина: Программирование на языках высокого уровня  Тема: Алгоритм Ахо-Корасик | | | |
| Студент группы 3331506/80401  Преподаватель | |  | А. А. Константинов  М. С. Ананьевский  «\_\_» \_\_\_\_\_\_\_\_ 2021 г. |
|  | Санкт-Петербург  2021 г | |  |

# Оглавление

[Оглавление 2](#_Toc69598965)

[Введение 3](#_Toc69598966)

[Принцип работы 4](#_Toc69598967)

[Оценка скорости и памяти 10](#_Toc69598968)

[Оценка скорости 10](#_Toc69598969)

[Оценка памяти 10](#_Toc69598970)

[Использование красно-чёрного дерева 11](#_Toc69598971)

[Анализ алгоритма 12](#_Toc69598972)

[Зависимость от размера словаря 12](#_Toc69598973)

[Зависимость от размера строки 13](#_Toc69598974)

[Список литературы 14](#_Toc69598975)

[Приложение 1 15](#_Toc69598976)

# 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. н. 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* попадается подстрока *aabaabac* — при считывании 5-го символа автомат потребуется 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 ссуфиксной 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 ссуфиксной 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.

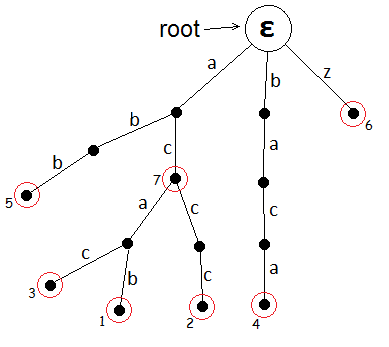


Рисунок 1 — Построение префиксного дерева

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

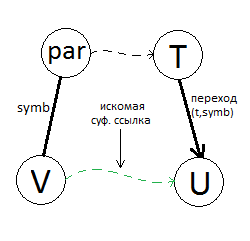


Рисунок 2 — Поиск суффиксной ссылки

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

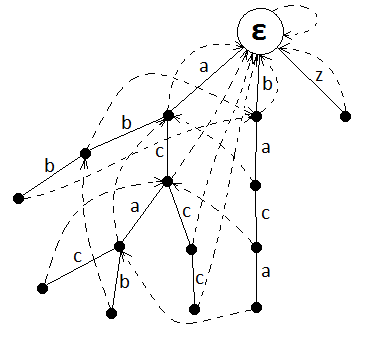


Рисунок 3 — Префиксное дерево с обозначенными суффиксными ссылками

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.

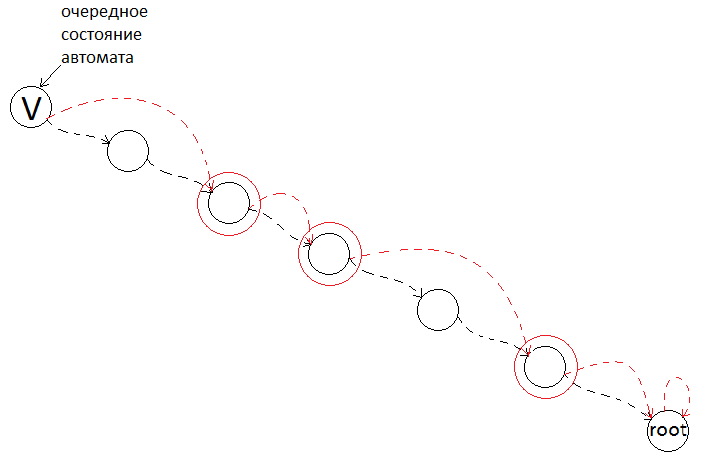


Рисунок 4 — Конечные суффиксные ссылки

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

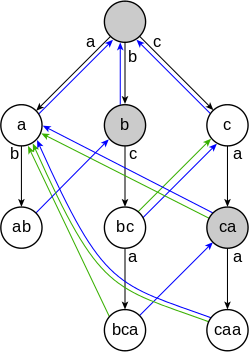


Рисунок 5 — Конечный автомат с показанными ссылками

# 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(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:

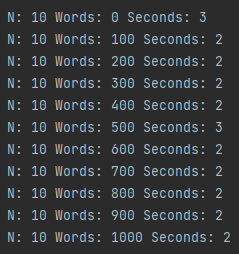


Рисунок 6 — Зависимость времени выполнения от размера словаря

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 подстрок, 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. https://www.geeksforgeeks.org/aho-corasick-algorithm-pattern-searching/

# Приложение 1

Код загловочного файла:

//  
// Created by alex on 15.04.2021.  
//  
  
#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;  
 int pattern\_number;  
 int suffix\_link;  
 map<char, int> move;  
 int suffix\_final\_link;  
 int parent;  
 char symbol;  
 bool isFinal;  
} Tree\_Vertex;  
  
class AC{  
public:  
 void AddString(const string &str);  
 AC();  
 ~AC()= default;  
 void Find(const string &str);  
private:  
 vector <Tree\_Vertex> tree;  
 vector <string> pattern;  
  
 static Tree\_Vertex MakeVertex(int parent, char symbol);  
 int GetSuffixLink(int vertex);  
 int GetMove(int vertex, char character);  
 int GetFinalSuffixLink(int vertex);  
 void Check(int vertex, int i);  
};  
  
#endif //AHO\_CORASICK\_AHO\_CORASICK\_H

Код *.cpp* файла:

//  
// Created by alex on 15.04.2021.  
//  
  
#include <iostream>  
#include "Aho\_Corasick.h"  
  
AC::AC() {  
 this->tree.push\_back(MakeVertex(0, '$'));  
}  
  
void AC::AddString(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(MakeVertex(num, (char) symbol));  
 this->tree[num].next\_vertex[symbol] = (int) this->tree.size() - 1;  
 }  
 num = this->tree[num].next\_vertex[symbol];  
 }  
 this->tree[num].isFinal = 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 = GetMove(u, (char) (str[i]));  
 this->Check(u, i + 1);  
 }  
}  
  
Tree\_Vertex AC::MakeVertex(int parent, char symbol) {  
 Tree\_Vertex vertex;  
 vertex.isFinal = false;  
 vertex.suffix\_link = -1;  
 vertex.suffix\_final\_link = -1;  
 vertex.parent = parent;  
 vertex.symbol = symbol;  
 return vertex;  
}  
  
int AC::GetSuffixLink(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 = GetMove(GetSuffixLink(this->tree[vertex].parent), this->tree[vertex].symbol);  
 }  
 return this->tree[vertex].suffix\_link;  
}  
  
int AC::GetMove(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] = GetMove(GetSuffixLink(vertex), (char) symbol);  
 }  
 }  
 return this->tree[vertex].move[symbol];  
}  
  
int AC::GetFinalSuffixLink(int vertex) {  
 if(this->tree[vertex].suffix\_final\_link == -1){  
 int u = GetSuffixLink(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 = GetFinalSuffixLink(u);  
 }  
 }  
 return this->tree[vertex].suffix\_final\_link;  
}  
  
void AC::Check(int vertex, int i) {  
 for(int u = vertex; u != 0; u = GetFinalSuffixLink(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;  
 }  
 }  
}