

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

А. А. Константинов

Преподаватель

М. С. Ананьевский

«__» _____ 2021 г.

Санкт-Петербург

2021 г

Оглавление

Оглавление	2
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. 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 потребует 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.

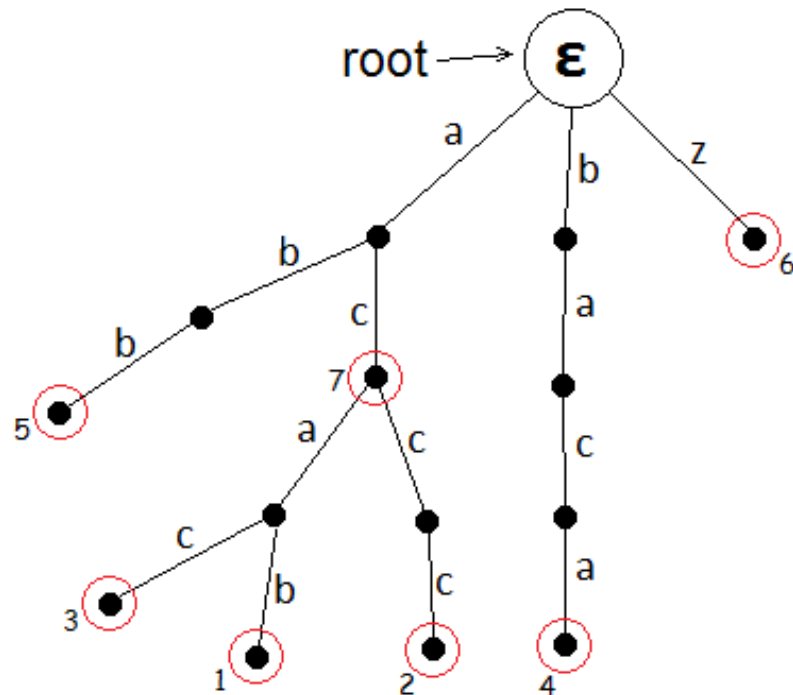


Рисунок 1 — Построение префиксного дерева

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

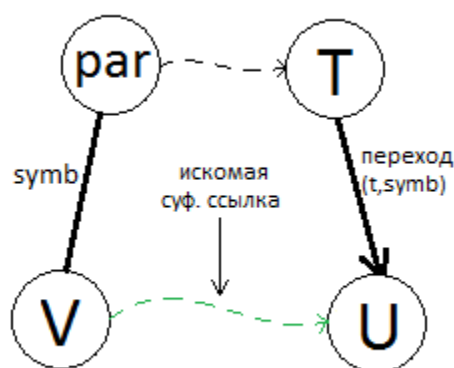


Рисунок 2 — Поиск суффиксной ссылки

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

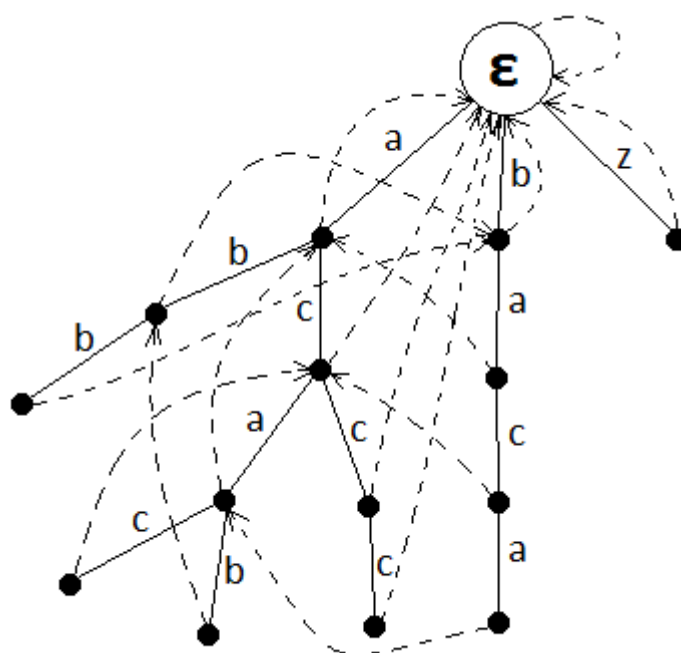


Рисунок 3 — Префиксное дерево с обозначенными суффиксными ссылками

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.

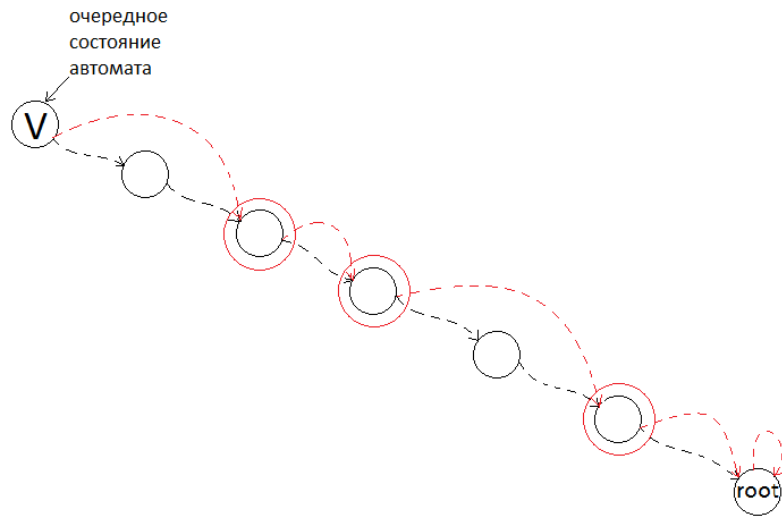


Рисунок 4 — Конечные суффиксные ссылки

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

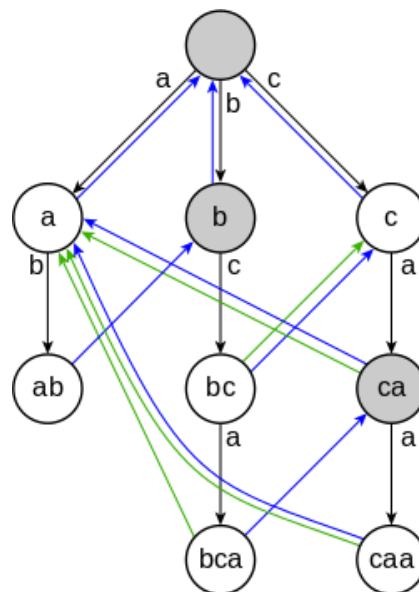


Рисунок 5 — Конечный автомат с показанными ссылками

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 $O(n * O(Check))$.

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.

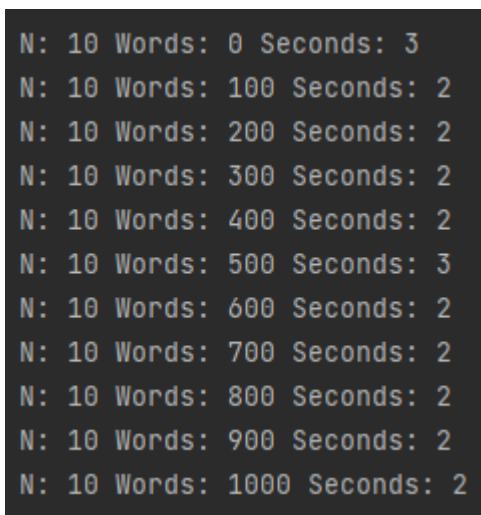
В качестве одной единицы входных данных использован текст книги «Над пропастью во ржи» Джерома Дэвида Сэлинджера. Размер книги составляет 386302 символа. В качестве словаря для поиска подстрок будут использоваться слова из самой книги, идущие в начале книги подряд, для простоты выбора.

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 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 заголовочного 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_Verx AC::MakeVertex(int parent, char symbol) {
    Tree_Verx 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;
        }
    }
}
}

```