**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра математического обеспечения и применения ЭВМ**

отчет

**по лабораторной работе №5**

**по дисциплине «Построение и анализ алгоритмов»**

**Тема: Алгоритм Ахо-Корасик**

|  |  |  |
| --- | --- | --- |
| Студентка гр. 7382 |  | Петрова А.С. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2019

**Задание.**

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T,1≤|T|≤100000 ).

Вторая - число n (1≤n≤3000), каждая следующая из n строк содержит шаблон из набора P={p1,…,pn}1≤|pi|≤75

Все строки содержат символы из алфавита {A, C, G, T, N}

Выход:

Все вхождения образцов из P в T.

Каждое вхождение образца в текст представить в виде двух чисел - i p

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p

(нумерация образцов начинается с 1).

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

Sample Input:

CCCA

1

CC

Sample Output:

1 1

2 1

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером.

В шаблоне встречается специальный символ, именуемого джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения Р в текст Т.

Например, образец аb??с? с джокером ? встречается дважды в тексте xabvccbababcax.

Символ джокер не входит в алфавит, символы которого используются в T. Каждый джокер соответствует одному символу, а не подстроке неопределенной длины. В шаблоне входит хотя бы один символ не джокер, те шаблоны вида ??? недопустимы.

Все строки содержат символы из алфавита {A,C,G,T,N}

Вход:

Текст (T,1≤|T|≤100000 )

Шаблон (P,1≤|P|≤40)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Sample Input:

ACT

A$

$

Sample Output:

1

**Вариант 2:**

Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

**Описание алгоритма:**

Алгоритм строит конечный автомат, которому затем передаёт строку поиска. Автомат получает по очереди все символы строки и переходит по соответствующим рёбрам. Если автомат пришёл в конечное состояние, соответствующая строка словаря присутствует в строке поиска.

Несколько строк поиска можно объединить в дерево поиска, так называемый бор (префиксное дерево). Бор является конечным автоматом, распознающим одну строку из m — но при условии, что начало строки известно.

Первая задача в алгоритме — научить автомат «самовосстанавливаться», если подстрока не совпала. При этом перевод автомата в начальное состояние при любой неподходящей букве не подходит, так как это может привести к пропуску подстроки (например, при поиске строки aabab, попадается aabaabab, после считывания пятого символа перевод автомата в исходное состояние приведёт к пропуску подстроки — верно было бы перейти в состояние a, а потом снова обработать пятый символ). Чтобы автомат самовосстанавливался, к нему добавляются суффиксные ссылки, нагруженные пустым символом ⌀ (так что детерминированный автомат превращается в недетерминированный). Например, если разобрана строка aaba, то бору предлагаются суффиксы aba, ba, a. Суффиксная ссылка — это ссылка на узел, соответствующий самому длинному суффиксу, который не заводит бор в тупик (в данном случае a).

Для корневого узла суффиксная ссылка — петля. Для остальных правило таково: если последний распознанный символ —c, то осуществляется обход по суффиксной ссылке родителя, если оттуда есть дуга, нагруженная символом c, суффиксная ссылка направляется в тот узел, куда эта дуга ведёт. Иначе — алгоритм проходит по суффиксной ссылке ещё и ещё раз, пока либо не пройдёт по корневой ссылке-петле, либо не найдёт дугу, нагруженную символом c.

Вычислительная сложность работы алгоритма зависит от организации данных. Например:

Если таблицу переходов автомата хранить как индексный массив — расход памяти O(nσ), вычислительная сложность O(nσ +H+k), где H — длина текста, в котором производится поиск, n — общая длина всех слов в словаре, σ — размер алфавита, k — общая длина всех совпадений. (В данной работе используется этот способ хранения).

Если таблицу переходов автомата хранить как красно-чёрное дерево — расход памяти снижается до O(n), однако вычислительная сложность поднимается до O((H+n)log σ +k).

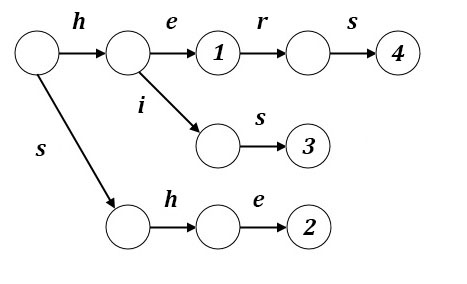
Шаг 1. Построение бора

Строим бор из строк.

Построение выполняется за время O(m), где m — суммарная длина строк.

Пример построенного бора

Бор для набора строк {he, she, his, hers}:



Шаг 2. Преобразование бора

Обозначим за [u] слово, приводящее в вершину u в боре.

Узлы бора можно понимать как состояния автомата, а корень как начальное состояние.

Узлы бора, в которых заканчиваются строки, становятся терминальными.

Для переходов по автомату заведём в узлах несколько функций:

parent(u) — возвращает родителя вершины u;

π(u)=δ(π(parent(u)),c) — суффиксная ссылка, и существует переход из parent(u) в u по символу c;

 — функция перехода.

Мы можем понимать рёбра бора как переходы в автомате по соответствующей букве. Однако одними только рёбрами бора нельзя ограничиваться. Если мы пытаемся выполнить переход по какой-либо букве, а соответствующего ребра в боре нет, то мы тем не менее должны перейти в какое-то состояние. Для этого нам и нужны суффиксные ссылки.

Суффиксная ссылка π(u)=v, если [v] — максимальный суффикс [u], [v]≠[u]. Функции перехода и суффиксные ссылки можно найти либо алгоритмом обхода в глубину с ленивыми вычислениями, либо с помощью алгоритма обхода в ширину.

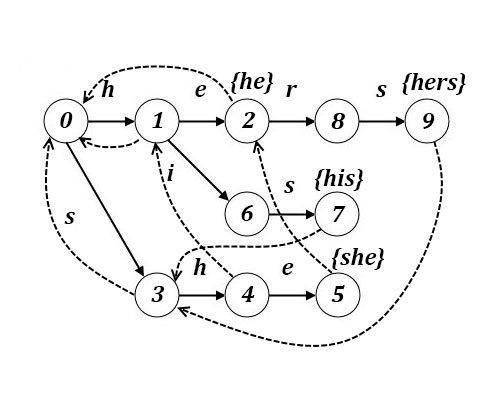
Из определений выше можно заметить два следующих факта:

функция перехода определена через суффиксную ссылку, а суффиксная ссылка — через функию переходов;

для построения суффиксных ссылок небходимо знать информацию только выше по бору от текущей вершины до корня.

Это позволяет реализовать функции поиска переходов по символу и суффиксных ссылок ленивым образом при помощи взаимной рекурсии.

Пример автомата Ахо-Корасик



Пунктиром обозначены суффиксные ссылки. Из вершин, для которых они не показаны, суффиксные ссылки идут в корень.

Суффиксная ссылка для каждой вершины u — это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине u. Единственный особый случай — корень бора: для удобства суффиксную ссылку из него проведём в себя же. Например, для вершины 5 с соответствующей ей строкой "she" максимальным подходящим суффиксом является строка "he". Видим, что такая строка заканчивается в вершине 2. Следовательно суффиксной ссылкой вершины для 5 является вершина 2 .

Шаг 3. Построение сжатых суффиксных ссылок

При построении автомата может возникнуть такая ситуация, что ветвление есть не на каждом символе. Тогда можно маленький бамбук заменить одним ребром. Для этого и используются сжатые суффиксные ссылки.



где up — сжатая суффиксная ссылка, т.е. ближайшее допускающее состояние (терминал) перехода по суффиксным ссылкам. Аналогично обычным суффиксным ссылкам сжатые суффиксные ссылки могут быть найдены при помощи ленивой рекурсии.

**Тестирование 1.**

|  |  |
| --- | --- |
| ACGTANCCGT  2  AC GT | 1 1  3 2  9 2  Vertex num: 5  arr size:3 |
| TANANTANTANAT  2  TAN NAT | 1 1  6 1  9 1  11 2  Vertex num: 7  arr size:4  TAN [8-10] cross NAT [10-12] |

**Тестирование 2.**

|  |  |
| --- | --- |
| ATNGTNACTNTCA  TN&  & | 2  5  9 |
| TCNGGNCTANCNTGCNT  A$  $ | 9 |

**ПРИЛОЖЕНИЕ А**

**Алгоритм Ахо\_Корасик.**

#include "pch.h"

#include <iostream>

#include <cstring>

#include <string>

#include <vector>

using namespace std;

// Размер алфавита

const int k = 60;

struct bohr\_vrtx

{

int next\_vrtx[k]; //next\_vrtx[i] — номер вершины, в которую мы придем по символу с номером i в алфавите

int pat\_num; //номер строки-образца, обозначаемого этой вершиной

int suff\_link; //suff\_link - суффиксная ссылка

int auto\_move[k]; //auto\_move - запоминание перехода автомата

int par; //par - вершина-отец в дереве

int64\_t suff\_flink; //suff\_flink - "хорошая" суф. ссылка

bool flag; //flag — бит, указывающий на то, является ли наша вершина исходной строкой

char symb; //символ на ребре от par к этой вершине

int count\_arc; //количество исхдящих дуг из вершины

};

class Bohr

{

vector<bohr\_vrtx> bohr;

vector<string> pattern;

vector<pair<int, int>> pattern\_entry\_index; // 1 - индекс вхождения, 2 - номер шаблона

//Функция добваления вершины в бор

bohr\_vrtx make\_bohr\_vrtx(int p, char c) //передаем номер отца и символ на ребре в боре

{

bohr\_vrtx v; //(255)=(2^8-1)=(все единицы в каждом байте памяти)=(-1 в дополнительном коде целого 4-байтного числа int)

memset(v.next\_vrtx, 255, sizeof(v.next\_vrtx));

memset(v.auto\_move, 255, sizeof(v.auto\_move));

v.flag = false; //По умолчанию: вершина - не сходная строка

v.suff\_link = -1; //изначально - суф. ссылки нет

v.par = p;

v.symb = c;

v.suff\_flink = -1;

v.count\_arc = 0;

return v;

}

public:

Bohr()

{

//добавляем единственную вершину - корень

bohr.push\_back(make\_bohr\_vrtx(0, '$'));

}

~Bohr() = default;

//Добавление строки в бор

void add\_string\_to\_bohr(const string &s)

{

int num = 0; //начинаем с корня

for (int i = 0; i < s.length(); i++)

{

int ch = s[i] - 'A'; //получаем номер в алфавите для символа строки

if (bohr[num].next\_vrtx[ch] == -1) //-1 - признак отсутствия ребра

{

bohr[num].count\_arc++;

bohr.push\_back(make\_bohr\_vrtx(num, ch)); //Создание вершины

bohr[num].next\_vrtx[ch] = bohr.size() - 1; //Установление ребра

}

num = bohr[num].next\_vrtx[ch]; //Переход в следующую вершину

}

bohr[num].flag = true; //Пометка конца паттерна

pattern.push\_back(s);

bohr[num].pat\_num = pattern.size() - 1;

}

private:

//Переход по суф. ссылке

int get\_suff\_link(int v)

{

if (bohr[v].suff\_link == -1) //если еще не считали

if (v == 0 || bohr[v].par == 0) //если v - корень или предок v - корень

bohr[v].suff\_link = 0;

else

bohr[v].suff\_link = get\_auto\_move(get\_suff\_link(bohr[v].par), bohr[v].symb);

return bohr[v].suff\_link;

}

//Функция автомата

int get\_auto\_move(int v, char ch)

{

if (bohr[v].auto\_move[ch] == -1) //Если для нашего случая переход автомата еще не определен, то определим его

if (bohr[v].next\_vrtx[ch] != -1) //Если из вершины v есть путь в ch

bohr[v].auto\_move[ch] = bohr[v].next\_vrtx[ch]; //Установим автомат перехода в нее

else if (v == 0) //Если пришли в корень

bohr[v].auto\_move[ch] = 0; //Установим автомат перехода в корень

else //Иначе

bohr[v].auto\_move[ch] = get\_auto\_move(get\_suff\_link(v), ch); //Установим автомат перехода рекурсивно в соответствии с автоматом для вершины в которую ведет суфф. ссылка текушей вершины

return bohr[v].auto\_move[ch];

}

// Нахождение "хорошей" суф. ссылки

int get\_suff\_flink(int v)

{

if (bohr[v].suff\_flink == -1)

{

int u = get\_suff\_link(v); //Переход по суфф. ссылке

if (u == 0) //либо v - корень, либо суфф. ссылка v указывает на корень

bohr[v].suff\_flink = 0;

else

bohr[v].suff\_flink = (bohr[u].flag) ? u : get\_suff\_flink(u); // В случае ненахождения конца паттерна, рекурсивный поиск

}

return bohr[v].suff\_flink;

}

// Проверка подстроки на нахождение в боре

void check(int v, int i)

{

for (int u = v; u != 0; u = get\_suff\_flink(u)) //Переходы по суфф. ссылкам

{

if (bohr[u].flag)

{

cout << i - pattern[bohr[u].pat\_num].length() + 1 << " " << bohr[u].pat\_num + 1 << endl;

pattern\_entry\_index.push\_back({ i - pattern[bohr[u].pat\_num].length(), bohr[u].pat\_num }); //Добавление в вектор для последующей обработки текста

}

}

}

public:

// Нахождение всех вхождений паттернов в текст

void find\_all\_pos(const string &s)

{

int u = 0;

for (int i = 0; i < s.length(); i++)

{

u = get\_auto\_move(u, s[i] - 'A'); //Переход автомата

check(u, i + 1); // Проверка подстроки на нахождение в боре

}

}

void print\_info()

{

cout << "Vertex num: " << bohr.size() << "\n";

cout << "arr size:" << pattern\_entry\_index.size() << "\n";

// вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска

while (pattern\_entry\_index.begin() != pattern\_entry\_index.end())

{

for (size\_t i = 1; i < pattern\_entry\_index.size(); i++)

{

if (pattern\_entry\_index[0].first + pattern[pattern\_entry\_index[0].second].size() > pattern\_entry\_index[i].first)

{

cout << pattern[pattern\_entry\_index[0].second] << " [" << pattern\_entry\_index[0].first << "-" << pattern\_entry\_index[0].first + pattern[pattern\_entry\_index[0].second].size() - 1 << "]"

<< " cross " << pattern[pattern\_entry\_index[i].second] << " [" << pattern\_entry\_index[i].first << "-" << pattern\_entry\_index[i].first + pattern[pattern\_entry\_index[i].second].size() - 1 << "]" << "\n";

}

}

pattern\_entry\_index.erase(pattern\_entry\_index.begin());

}

}

};

int main()

{

string T, pattern;

int n;

cin >> T >> n;

// Инициализация бора

Bohr bohr;

for (int i = 0; i < n; i++)

{

cin >> pattern;

// Добавление паттернов в бор

bohr.add\_string\_to\_bohr(pattern);

}

// Нахождение всех паттернов в строке T

bohr.find\_all\_pos(T);

bohr.print\_info();

system("pause");

}

**ПРИЛОЖЕНИЕ Б**

**Прогрмма с джокером.**

#include "pch.h"

#include <iostream>

#include <cstring>

#include <string>

#include <vector>

using namespace std;

// Размер алфавита

const int k = 90;

struct bohr\_vrtx

{

int next\_vrtx[k]; //next\_vrtx[i] — номер вершины, в которую мы придем по символу с номером i в алфавите

int pat\_num; //номер строки-образца, обозначаемого этой вершиной

int suff\_link; //suff\_link - суффиксная ссылка

int auto\_move[k]; //auto\_move - запоминание перехода автомата

int par; //par - вершина-отец в дереве

int64\_t suff\_flink; //suff\_flink - "хорошая" суф. ссылка

bool flag; //flag — бит, указывающий на то, является ли наша вершина исходной строкой

char symb; //символ на ребре от par к этой вершине

int count\_arc; //количество исхдящих дуг из вершины

};

class Bohr

{

vector<bohr\_vrtx> bohr;

vector<string> pattern;

vector<pair<int, int>> remove\_text;

public:

char joker;

private:

//Функция добваления вершины в бор

bohr\_vrtx make\_bohr\_vrtx(int p, char c) //передаем номер отца и символ на ребре в боре

{

bohr\_vrtx v; //(255)=(2^8-1)=(все единицы в каждом байте памяти)=(-1 в дополнительном коде целого 4-байтного числа int)

memset(v.next\_vrtx, 255, sizeof(v.next\_vrtx));

memset(v.auto\_move, 255, sizeof(v.auto\_move));

v.flag = false; //По умолчанию: вершина - не сходная строка

v.suff\_link = -1; //изначально - суф. ссылки нет

v.par = p;

v.symb = c;

v.suff\_flink = -1;

v.count\_arc = 0;

return v;

}

public:

Bohr()

{

//добавляем единственную вершину - корень

bohr.push\_back(make\_bohr\_vrtx(0, '$'));

}

//Добавление строки в бор

void add\_string\_to\_bohr(const string &s)

{

int num = 0; //начинаем с корня

for (int i = 0; i < s.length(); i++)

{

int ch = s[i] - ' '; //получаем номер в алфавите для символа строки

if (bohr[num].next\_vrtx[ch] == -1) //-1 - признак отсутствия ребра

{

bohr[num].count\_arc++;

bohr.push\_back(make\_bohr\_vrtx(num, ch)); //Создание вершины

bohr[num].next\_vrtx[ch] = bohr.size() - 1; //Установление ребра

}

num = bohr[num].next\_vrtx[ch]; //Переход в следующую вершину

}

bohr[num].flag = true; //Пометка конца паттерна

pattern.push\_back(s);

bohr[num].pat\_num = pattern.size() - 1;

}

private:

//Переход по суф. ссылке

int get\_suff\_link(int v)

{

if (bohr[v].suff\_link == -1)

if (v == 0 || bohr[v].par == 0)

bohr[v].suff\_link = 0;

else

bohr[v].suff\_link = get\_auto\_move(get\_suff\_link(bohr[v].par), bohr[v].symb);

return bohr[v].suff\_link;

}

//Функция автомата

int get\_auto\_move(int v, char ch)

{

if (bohr[v].auto\_move[ch] == -1)

if (bohr[v].next\_vrtx[ch] != -1)

bohr[v].auto\_move[ch] = bohr[v].next\_vrtx[ch];

else if (bohr[v].next\_vrtx[joker] != -1)

bohr[v].auto\_move[ch] = bohr[v].next\_vrtx[joker];

else if (v == 0)

bohr[v].auto\_move[ch] = 0;

else

bohr[v].auto\_move[ch] = get\_auto\_move(get\_suff\_link(v), ch);

return bohr[v].auto\_move[ch];

}

// Нахождение "хорошей" суф. ссылки

int get\_suff\_flink(int v)

{

if (bohr[v].suff\_flink == -1)

{

int u = get\_suff\_link(v);

if (u == 0)

bohr[v].suff\_flink = 0;

else

bohr[v].suff\_flink = (bohr[u].flag) ? u : get\_suff\_flink(u);

}

return bohr[v].suff\_flink;

}

// Проверка подстроки на нахождение в боре

void check(int v, int i)

{

for (int u = v; u != 0; u = get\_suff\_flink(u))

{

if (bohr[u].flag)

{

cout << i - pattern[bohr[u].pat\_num].length() + 1 << endl;

remove\_text.push\_back({ i - pattern[bohr[u].pat\_num].length(), pattern[bohr[u].pat\_num].size() });

}

}

}

public:

// Нахождение всех вхождений паттернов в текст

void find\_all\_pos(const string &s)

{

int u = 0;

for (int i = 0; i < s.length(); i++)

{

u = get\_auto\_move(u, s[i] - ' ');

check(u, i + 1);

}

}

};

int main()

{

string T, pattern;

char joker;

cin >> T >> pattern >> joker;

joker -= ' ';

// Инициализация бора

Bohr bohr;

//Инициализация джокера

bohr.joker = joker;

bohr.add\_string\_to\_bohr(pattern);

// Нахождение всех паттернов в строке T

bohr.find\_all\_pos(T);

system("pause");

}