**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №5**

**по дисциплине «Построение и Анализ алгоритмов» Тема: Поиск образца в тексте. Алгоритм Рабина-Карпа.**

Студентка гр. 1383 Федорова О.В.



Преподаватель Иванов Д.В.



Санкт-Петербург

2023

**Цель работы.**

Ознакомиться с алгоритмом поиска подстрок в строке с использованием алгоритма Карпа-Рабина. Реализовать программу, в которой используется данный алгоритм.

**Задание**

Разработайте программу, решающую задачу точного поиска набора образцов.

**Вход:**

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

Вторая - число nn (1≤n≤30001≤n≤3000), каждая следующая из nn строк содержит шаблон из набора P={p1,…,pn}1≤∣pi∣≤75P={p1​,…,pn​}1≤∣pi​∣≤75

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

**Выход:**

Все вхождения образцов из PP в TT.

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

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

(нумерация образцов начинается с 1).

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

**Sample Input:**

NTAG

3

TAGT

TAG

T

**Sample Output:**

2 2

2 3

**Вариант 1**. На месте джокера может быть любой символ, за исключением заданного.

**Ход работы:**

Для выполнения работы были созданы следующие структуры и функции:

1. Структура Vertex представляет вершину в боре (trie) для алгоритма Ахо-Корасик. Она содержит следующие поля:
   * next\_vertex[k]: массив, где каждый элемент указывает на следующую вершину по соответствующей букве алфавита.
   * pat\_num: номер строки-образца, если вершина является терминальной (концом образца).
   * terminal: флаг, указывающий, является ли вершина терминальной.
   * suffix\_link: ссылка на вершину, образующую наибольший суффикс данного образца в боре.
   * auto\_move[k]: массив, хранящий результаты автоматического перехода из данной вершины.
   * par: вершина-отец в дереве.
   * symb: символ на ребре от par к этой вершине.
   * suff\_flink: ссылка на вершину, образующую наибольший суффикс, который также является терминальным образцом.
2. std::vector<Vertex> bohr - вектор, содержащий все вершины бора.
3. std::vector<std::string> pattern - вектор, содержащий образцы (строки), которые нужно найти в тексте.

Функции и процедуры:

1. make\_bohr\_vertex - создает новую вершину бора с заданными параметрами.
2. bohr\_ini - инициализирует бор, добавляя корневую вершину.
3. add\_str\_to\_bohr - добавляет строку в бор.
4. is\_string\_in\_bohr - проверяет, содержится ли данная строка в боре.
5. get\_auto\_move - вычисляет результат автоматического перехода из данной вершины по заданному символу.
6. get\_suff\_link - вычисляет ссылку на вершину, образующую наибольший суффикс данной вершины в боре.
7. get\_suff\_flink - вычисляет ссылку на вершину, образующую наибольший суффикс, который также является терминальным образцом.
8. check - проверяет наличие терминальных образцов, начиная с данной вершины.
9. find\_all\_pos - находит все вхождения образцов в заданном тексте.
10. cmp - функция сравнения для сортировки результатов поиска.

Для демонстрации работы программ были созданы функции jock\_var() и jock\_var() для выполнения 2 и 1 задания на Stepik. В 1 задании в бор последовательно добавляются строки-паттерны в бор, после чего запускается функция find\_all\_pos.

Для выполнения второго задания и джокером, в момент получения строки-паттерна и символа-джокера, программа в цикле добавляет в бор строки, в которых на месте джокера стоит любой символ из алфавита alpha. Для оптимизации работы программы, массив alpha был урезан до 5, так же создан массив alpha\_num для хранения индекса в массиве alpha символа char.

Для выполнения задания по варианту, массив для алфавита был расширен. В качестве символов могут выступать любые символы с кодами от 33 до 127. Вместо того, чтобы записывать номера символов в алфавите в массив, можно теперь просто хранить код стартовой вершины(33) и вычитать ее из кода очередного символа, получив тем самым номер в алфавите.

После считывания строки паттерна и джокера, на место символа-джокера в строку паттерн подставляются все символы алфавита, кроме джокера, после чего запускается поиск.

Выводы:

Изучен алгоритм Ахо-Корасика и реализована программа, демонстрирующая его работу.

**ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Название файла: my\_lb4.cpp

#include <iostream>

#include "vector"

#include "map"

#include "string"

#include <cstring>

#include "algorithm"

const int k = 5;

std::vector <int> alpha\_num = **{**0,-1,1,-1,-1,-1,2,-1,-1,2,-1,-1,-1,4,-1,-1,-1,-1,-1,3,-1,-1,-1,-1,-1,-1**}**;

std::vector <char> alpha = **{**'A', 'C', 'G', 'T', 'N'**}**;

std::map <char, int> gen;

char sym\_start = 'A';

struct Vertex {

int next\_vertex[k];

int pat\_num;

bool terminal = false;

int suffix\_link;

int auto\_move[k];

int par; //auto\_move - запоминание перехода автомата, par - вершина-отец в дереве

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

int suff\_flink;

};

std::vector <Vertex> bohr;

std::vector <std::string> pattern;

Vertex make\_bohr\_vertex(int p, char c){

Vertex v;

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

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

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

v.terminal = false;

v.suffix\_link = -1;

v.suff\_flink = -1;

v.par = p;

v.symb = c;

return v;

}

void bohr\_ini(){

//добавляем единственную вершину - корень

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

}

void add\_str\_to\_bohr(std::string s){

int num=0; //начинаем с корня

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

char ch = alpha\_num[s[i]- sym\_start]; //получаем номер в алфавите

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

bohr.push\_back(make\_bohr\_vertex(num, ch));

bohr[num].next\_vertex[ch] = bohr.size()-1;

}

num=bohr[num].next\_vertex[ch];

}

bohr[num].terminal = true;

pattern.push\_back(s);

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

}

bool is\_string\_in\_bohr(std::string s){

int num=0;

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

char ch = alpha\_num[s[i] - sym\_start];

if (bohr[num].next\_vertex[ch]==-1){

return false;

}

num = bohr[num].next\_vertex[ch];

}

return true;

}

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

int get\_suff\_link(int v){

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

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

bohr[v].suffix\_link=0;

else

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

return bohr[v].suffix\_link;

}

int get\_suff\_flink(int v){

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

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

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

bohr[v].suff\_flink=0;

else

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

}

return bohr[v].suff\_flink;

}

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

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

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

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

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];

}

std::vector <std::pair <int, int>> ans;

void check(int v,int i){

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

if (bohr[u].terminal)

ans.push\_back(std::pair<int, int> (i-pattern[bohr[u].pat\_num].length()+1, bohr[u].pat\_num + 1));

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

}

}

void find\_all\_pos(std::string s){

int u=0;

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

u=get\_auto\_move(u,alpha\_num[s[i]- sym\_start]);

check(u,i+1);

}

}

bool cmp(std::pair <int, int> & p1, std::pair <int, int> & p2) {

if(p1.first < p2.first) return true;

if(p1.first == p2.first && p1.second < p2.second) return true;

return false;

}

void create\_bohr(std::string tmp, int pos, char jock) {

//std::cout<<"Creating"<<std::endl;

if(pos == tmp.length()) add\_str\_to\_bohr(tmp);

else if(tmp[pos] != jock) create\_bohr(tmp, pos+1, jock);

else if(tmp[pos] == jock) {

for(auto i : gen) {

tmp[pos] = i.first;

create\_bohr(tmp, pos + 1, jock);

}

}

}

void rec\_create(std::string tmp, int pos, char jock) {

if(pos == tmp.size() - 1) add\_str\_to\_bohr(tmp);

else if(tmp[pos] != jock) rec\_create(tmp, pos + 1, jock);

else if(tmp[pos] == jock) {

for (auto sym: alpha) {

if(sym != jock) {

tmp[pos] = sym;

rec\_create(tmp, pos + 1, jock);

}

}

}

}

void bohr\_var(){

std::string txt;

std::cin>>txt;

int n;

std::cin>>n;

bohr\_ini();

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

std::string tmp;

std::cin>>tmp;

add\_str\_to\_bohr(tmp);

}

find\_all\_pos(txt);

std::sort(ans.begin(), ans.end(), cmp);

for(auto i : ans)

std::cout<<i.first<<" "<<i.second<<std::endl;

}

void jock\_var(){

std::string txt;

std::cin>>txt;

for(int i = 0; i < txt.size(); i++) {

if(gen.find(txt[i]) == gen.end()) {

gen.insert({txt[i], gen.size()});

}

}

bohr\_ini();

std::string tmp;

char jock;

std::cin>>tmp>>jock;

create\_bohr(tmp, 0, jock);

find\_all\_pos(txt);

for(auto i : ans)

std::cout<<i.first<<std::endl;

}

int main() {

//bohr\_var();

//jock\_var();

return 0;

}