

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

Студент гр. 8304

Бочаров Ф.Д.

Преподаватель

Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

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

Задание.

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

Например, образец $ab??c?ab??c?$ с джокером $??$ встречается дважды в тексте $xabvsscbaababcsaxxabvsscbaababcsax$.

Символ джокер не входит в алфавит, символы которого используются в T . Каждый джокер соответствует одному символу, а не подстроке неопределённой длины. В шаблон входит хотя бы один символ не джокер, т.е. шаблоны вида $???$ недопустимы.

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

Вариант 2.

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

Описание алгоритма.

По полученным образцам строим бор. Построение выполняется за время $O(m)$, где m — суммарная длина строк.

Узлы бора можно понимать как состояния автомата, а корень как начальное состояние. Узлы бора, в которых заканчиваются строки, становятся терминальными. 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 нужны суффиксные ссылки.

При построении автомата может возникнуть такая ситуация, что ветвление есть не на каждом символе. Тогда можно использовать сжатые суффиксные ссылки, т.е. ближайшее допускающее состояние (терминал) перехода по суффиксным ссылкам.

По очереди просматриваем символы текста. Для очередного символа s переходим из текущего состояния u в состояние, которое вернёт функция $\delta(u, s)$. Оказавшись в новом состоянии, отмечаем по сжатым суффиксным ссылкам строки, которые нам встретились и их позицию (если требуется). Если новое состояние является терминалом, то соответствующие ему строки тоже отмечаем.

Для того чтобы найти все вхождения в текст заданного шаблона с масками Q , необходимо обнаружить вхождения в текст всех его безмасочных кусков. Пусть $\{Q_1, \dots, Q_k\}$ — набор подстрок Q , разделённых масками, и пусть $\{l_1, \dots, l_k\}$ — их стартовые позиции в Q . Например, шаблон $ab\phi fc\phi$ содержит две подстроки без масок ab и c и их стартовые позиции соответственно 1 и 5. Для алгоритма понадобится массив C . $C[i]$ — количество встретившихся в тексте безмасочных подстрок шаблона, который начинается в тексте на позиции i . Тогда появление подстроки Q_i в тексте на позиции j будет означать возможное появление шаблона на позиции $j - l_i + 1$.

1. Используя алгоритм Ахо-Корасик, находим безмасочные подстроки шаблона Q : когда находим Q_i в тексте T на позиции j , увеличиваем на единицу $C[j - l_i + 1]$.
2. Каждое i , для которого $C[i] = k$, является стартовой позицией появления шаблона Q в тексте.

Рассмотрим подстроку текста $T[i \dots i+n-1]$. Равенство $C[i] = k$ будет означать, что подстроки текста $T[i+l_1 \dots i+l_1+|Q_1|-1]$, $T[i+l_2 \dots i+l_2+|Q_2|-1]$ и так далее будут равны соответственно безмасочным подстрокам шаблона $\{Q_1, \dots, Q_k\}$. Остальная часть шаблона является масками, поэтому шаблон входит в текст на позиции i .

Поиск подстрок заданного шаблона с помощью алгоритма Ахо-Корасик выполняется за время $O(m+n+a)$, где

n — суммарная длина подстрок, то есть длина шаблона,

m — длина текста,

a — количество появлений подстрок шаблона.

Далее просто надо пробежаться по массиву S и просмотреть значения ячеек за время $O(m)$.

Описание основных структур данных и функций.

```
struct bohr_vrtx {
    bohr_vrtx(int p, char c) {
        this->par = p;
        this->symb = c;
    }
    int next_vrtx[k] = { -1, -1, -1, -1, -1 };
    std::vector<int> pat_num;
    int suff_link = -1;
    int auto_move[k] = { -1, -1, -1, -1, -1 };
    int par = -1;
    int suff_flink=-1;
    bool flag = false;
    char symb = 0;
};
```

- структура для хранения вершины бора, где

$next_vrtx[i]$ — номер вершины, в которую мы придем по символу с номером i в алфавите.

$flag$ — флаг, проверяющий, является ли наша вершина исходной строкой.

pat_num — номера строк-образцов, обозначаемых этой вершиной.

$suff_link$ — суффиксная ссылка.

$auto_move$ — запоминание перехода автомата.

par — вершина-отец в дереве.

$symb$ — символ на ребре от par к этой вершине.

$suff_flink$ — сжатая суффиксная ссылка.

```
void add_str_to_bohr(std::string& s);
```

- добавления строки в бор.

```
void find_all (std::istream& input, std::ostream& output);
```

- поиск введенных образцов в введенном тексте.

```
void check(int v, int i, std::vector<int>& C, std::ostream& output);
```

- передвижение по сжатым суффиксным ссылкам.

```
int get_suf_flink(int v);
```

- вычисления сжатой суффиксной ссылки.

```
int get_move(int v, char ch);
```

- переход по состояниям автомата. Принимает текущее состояние и символ, по которому осуществляется переход. Возвращает новое состояние.

```
int get_suf_link(int v);
```

- вычисление суффиксной ссылки.

Тестирование.

Таблица 1 – Результат работы.

Ввод	Вывод
ACTANCA A??A? ?	1
AFGBNSFS G?N	3
ACABA A?A	1 3
ACGACTACNACG AC*AC	1 4 7

Вывод.

В ходе выполнения работы, были получены знания о структуре данных бор, алгоритме Ахо-Корасик. Была написана программа поиска вхождения образца в строку.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД

```
#include <iostream>
#include <string>
#include <map>
#include <vector>
#include <fstream>

std::map<char, int> alphabet;
const int k = 5;
std::vector<int> pos;

struct bohr_vrtx {
    bohr_vrtx(int p, char c) {
        this->par = p;
        this->symb = c;
    }
    int next_vrtx[k] = { -1,-1,-1,-1,-1 };
    std::vector<int> pat_num;
    int suff_link = -1;
    int auto_move[k] = { -1,-1,-1,-1,-1 };
    int par = -1;
    int suff_flink = -1;
    bool flag = false;
    char symb = 0;
};

std::vector<bohr_vrtx> bohr = { bohr_vrtx(-1, 0) };
std::vector<std::string> pattern;

int get_move(int v, char ch);

int get_suf_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_move(get_suf_link(bohr[v].par), bohr[v].symb);
    return bohr[v].suff_link;
}

int get_move(int cur, char ch) { // перехода по состояниям автомата
    if (bohr[cur].auto_move[ch] == -1)
        if (bohr[cur].next_vrtx[ch] != -1) //если есть ребро
            bohr[cur].auto_move[ch] = bohr[cur].next_vrtx[ch]; //то пройдем по нему
        else
            if (cur == 0)
                bohr[cur].auto_move[ch] = 0;
            else //иначе идем по суфф ссылке
                bohr[cur].auto_move[ch] = get_move(get_suf_link(cur), ch);
    return bohr[cur].auto_move[ch];
}

int get_suf_flink(int v) { //вычисление хорошей суфф ссылки
    if (bohr[v].suff_flink == -1) {
        int u = get_suf_link(v);
        if (u == 0)
            bohr[v].suff_flink = 0;
        else
            bohr[v].suff_flink = (bohr[u].flag) ? u : get_suf_flink(u);
    }
}
```

```

        return bohr[v].suff_flink;
    }

void check(int v, int i, std::vector<int>& C, std::ostream& output) { //хождение по хорошим
супф ссылкам
    for (int u = v; u != 0; u = get_suf_flink(u)) {
        if (bohr[u].flag) {
            for (int k : bohr[u].pat_num) {
                int j = i - pattern[k].size() + 1;
                int che = j - pos[k] + 1;
                if (che > 0 && che < C.size()) { //проверка выхода за рамки
                    ++C[che]; //увеличиваем значение
                }
            }
        }
    }
}

void add_str_to_bohr(std::string& s) { //добавление строки в бор
    int num = 0;
    for (char i : s) { //проходим по всей строке
        char ch = alphabet[i]; //получаем номер элемента в алфавите
        if (bohr[num].next_vrtx[ch] == -1) { //если нет ребра
            bohr.push_back(bohr_vrtx(num, ch)); //добавляем его
            bohr[num].next_vrtx[ch] = bohr.size() - 1;
        }
        num = bohr[num].next_vrtx[ch];
    }
    bohr[num].flag = true;
    pattern.push_back(s);
    bohr[num].pat_num.push_back(pattern.size() - 1);
}

void find_all(std::istream& input, std::ostream& output) { //функция поиска
    std::string T;
    std::string temp;
    std::string cur;
    char joker ;
    input >> T >> temp >> joker ;
    for (int i = 0; i < temp.size(); ++i) { //разделение образца на куски
        if (temp[i] == joker) {
            if (cur.empty())
                continue;
            else {
                output << "Found substring: " << cur << std::endl;
                add_str_to_bohr(cur); // добавляем в бор
                pos.push_back(i - cur.size()); // записываем индекс
                cur = "";
                continue;
            }
        }
        else if (i == temp.size() - 1) {
            cur += temp[i]; // добавляем в бор
            output << "Found substring: " << cur << std::endl;
            add_str_to_bohr(cur); // оставшийся кусок
            pos.push_back(i - cur.size() + 1);
        }
        cur += temp[i];
    }
    output << std::endl;
    int v = 0;
    std::vector<int> F(T.size()); //вектор для хранения количества подстрок без джокера
    output << "State changes:" << std::endl;
    for (int i = 0; i < T.length(); i++) {

```



```

        output << "From " << v;
        v = get_move(v, alphabet[T[i]]);
        output << " to " << v << std::endl;
        check(v, i, F, output);
    }
    std::vector<int> inter; //вектор для хранения ответов
    for (int k = 0; k < F.size(); ++k)
        if (F[k] == pos.size()) {
            if (k + temp.size() - 1 <= T.size()) {
                output << std::endl;
                output << "Found result at " << k << " position" << std::endl
                    << "Result is \"" << T.substr(k - 1, temp.size()) << "\""
<< std::endl;

                for (int h : inter) {
                    if ((k - h) < temp.size()) {
                        output << "\"" << T.substr(h - 1, temp.size()) <<
<< " from pos = " << h << " has
intersection with " << "\""
<< T.substr(k - 1, temp.size()) << "\"" <<
" from pos = "
<< k << std::endl; //то значит есть
пересечение
                    }
                }
                inter.push_back(k); //вносим ответ в вектор
            }
        }
    output << std::endl;
    output << "Coincidences was found at: "; //вывод индексов
    for (int res : inter) //совпадений
        output << res << " ";
    output << std::endl << std::endl;
    output << "Number of vertices: " << bohr.size() << std::endl; //количество вершин
}

int main() {
    alphabet['A'] = 0; //инициализация алфавита
    alphabet['C'] = 1;
    alphabet['G'] = 2;
    alphabet['T'] = 3;
    alphabet['N'] = 4;

    find_all(std::cin, std::cout);

    return 0;
}

```