**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 набора подстрок в строке (Алгоритм Ахо-Корасик)**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3342 |  | Песчатский С. Д. |
| Преподаватель |  | Виноградова Е. В. |

Санкт-Петербург

2025

## 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}{*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:**

NTAG

3

TAGT

TAG

T

**Sample Output:**

2 2

2 3

Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с *джокером*.  
  
В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P*P* необходимо найти все вхождения РР в текст ТТ.  
  
Например, образец аb??с?а*b*??с? с джокером ?? встречается дважды в тексте xabvccbababcax*xabvccbababcax*.  
  
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}  
  
**Вход:**  
Текст (T,1≤∣T∣≤100000*T*)  
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:**

ACTANCA

A$$A$

$

**Sample Output:**

1

Индивидуальный вариант: 7. Вывод графического представления автомата.

**Описание алгоритма**

Алгоритм Ахо-Корасика — это производительный метод поиска набора подстрок (шаблонов) в тексте. Он развивает идеи алгоритмов Кнута-Морриса-Пратта и Бойера-Мура, используя конечный автомат для оперативного обнаружения совпадений.

Шаги алгоритма:

1. Формирование префиксного дерева (Trie)

Изначально строится дерево префиксов (Trie), куда последовательно добавляются все строки-образцы. Корневая вершина остается пустой. Каждый узел соответствует символу, а цепочка от корня до узла образует начало одной или нескольких строк. Конечные узлы маркируют завершение образцов. Если символ присутствует в текущей ветви, алгоритм переходит по существующей связи; в противном случае создается новый узел.

2. Суффиксные ссылки (failure links)

Суффиксные ссылки обеспечивают эффективную обработку расхождений при анализе текста, позволяя мгновенно переключаться на альтернативные варианты продолжения поиска. Эти элементы критически важны для работы автомата, минимизируя количество избыточных операций сравнения.

Формирование суффиксных ссылок:

После построения префиксного дерева (Trie) выполняется последовательная обработка всех узлов для установки ссылок, указывающих на максимально длинный суффикс, совпадающий с началом другого шаблона.

Принципы построения:

- Корневая вершина ссылается на себя

- Узлы первого уровня направляют свои суффиксные ссылки на корень

- Для остальных элементов ссылки определяются путем анализа цепочки родительских связей

Процедура создания ссылок:

1. Выполняется обход дерева в ширину (BFS), начиная с дочерних элементов корня

2. Для каждого узла:

- Определяется родительский элемент и его суффиксная ссылка

- Осуществляется поиск соответствующего суффикса в структуре Trie

- При обнаружении совпадения фиксируется ссылка, иначе поиск продолжается вверх по иерархии

3. Процесс повторяется для всех вершин дерева

Выходные ссылки (output links)

Конечные узлы (маркирующие завершение образцов) регистрируются в специальном списке. Дополнительно для каждого узла формируется выходная ссылка на ближайший терминальный узел в цепочке суффиксов.

Поиск совпадений в тексте:

1. Последовательный анализ текста посимвольно

2. При наличии соответствия в Trie - переход по соответствующей дуге

3. При отсутствии соответствия - использование суффиксной ссылки

4. Фиксация всех обнаруженных образцов при достижении конечных узлов

5. Дополнительная проверка по выходным ссылкам

Вычислительная сложность:

Временная сложность поиска составляет O(n+m+z), где:

- n - размер входного текста

- m - общая длина всех шаблонов

- z - количество обнаруженных совпадений

Особенности работы:

Алгоритм способен находить множественные вхождения шаблонов в одной позиции текста. Например, в строке "sheisashers" при поиске ["she","he","his","hers"] могут быть выявлены несколько соответствий одновременно.

Автомат с суффиксными переходами гарантирует линейное время поиска, однако общее количество обнаруженных совпадений может существенно влиять на производительность.

Требования к памяти:

Объем используемой памяти оценивается как O(m), что обусловлено необходимостью хранения:

- Всех узлов префиксного дерева

- Суффиксных и выходных ссылок

- Дополнительных структур данных

**Описание функций и структур данных**

1. struct Node Внутренний класс, представляющий узел в автомате Ахо-Корасик. Каждый узел содержит следующие поля:

* children: map<char, node\*> - переходы к другим узлам по символам.
* fail: node\* - суффиксная ссылка, указывающая на следующий возможный узел в случае несоответствия.
* dict: node\* - словарная ссылка, указывающая на узел, содержащий найденное слово.
* pattern\_info: vector<pair<int, int>> - индекс паттерна и его сдвиг
* word: string - слово, заканчивающееся в данном узле (если узел является концом шаблона).
* id: string - уникальный идентификатор узла.
* patternIndex: int - индекс шаблона, если узел является концом шаблона.
* toString(): string – метод возвращающий информацию об узле в виде строки

1. class AhoCorasick Основной класс, реализующий алгоритм Ахо-Корасик для эффективного поиска множества строк в тексте. Содержит следующие поля и методы:

* root: Node\* - корневой узел дерева
* subpatterns: vector<pair<string, int>> - подпаттерны и их сдвиг
* pattern\_len: int – длина исходного текста
* void buildTrie() Построение префиксного дерева (бора) из набора шаблонов.
* void SpecbuildTrie() Построение префиксного дерева (бора) из набора шаблонов, вариация для шаблона с джокером.
* void buildFailureLInks() создание суффискных:
* void Aho(vector<string> pats) – создание дерева для обычной работы
  + pats: vector<string> - набор шаблонов
* void SpecAho(const vector<pair<string, int>>& subpats, int len) – создание дерева для поиска шаблона с джокером
  + subpats: const vector<pair<string, int>>& - шаблон с джокером, разбитый на подшаблоны
  + len: int – длина изначального шаблона
* vector<pair<int, int>> search(string text) – ход алгоритма для поиска вхождений каждого шаблона в текст, возвращающий индекс вхождения и номер шаблона
  + text: string – текст, в котором ищутся вхождения
* vector<int> Specsearch(const string& text) – ход алгоритма для поиска вхождений шаблона в текст, возвращающий индекс вхождения, используется для случая с джокером
  + text: string – текст, в котором ищутся вхождения
* void visualize() – визуализация обычной работы алгоритма
* void SpecVisualize() – визуализация работы алгоритма, для случая с джокером
* void generateDotFile(const string& filename) – создания dot файла для отрисовки созданного дерева
  + filename: const string& - название файла
* generateAndShowGraph() – генерация изображения, по созданному dot файлу
* vector<pair<string, int>> split\_pattern(const string& pattern, char wild) – разбиение шаблона с джокером на подшаблоны, без учёта джокера. Возвращает массив пар подстроки и её сдвига
  + patter: const string& - шаблон
  + wild: char - символ джокера
* int main() – головная функция программы, где происходит считывание строк.

**Тестирование**

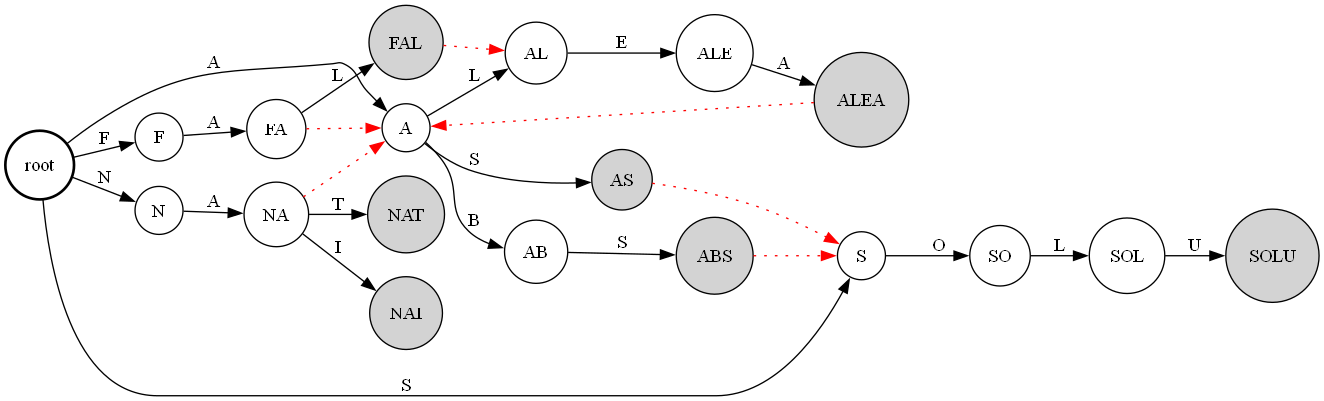
Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

|  |  |  |  |
| --- | --- | --- | --- |
| № п/п | Входные данные | Выходные данные | Комментарии |
|  | ABSOLUTEASSASSNIATIONOFALEADER  7  ABS  AS  NAI  NAT  ALEA  FAL  SOLU | 1 1  3 7  9 2  12 2  23 6  24 5 | Верный вывод |
|  | ntag  3  tagt  tag  t | 2 2  2 3 | Верный вывод |
|  | TTNACGNTTNACGN  ?N?CG?  ? | 2  9 | Верный вывод |
|  | ACGT  ????  ? | -1 | Верный вывод |

**Графическое представление автомата**

По индивидуальному варианту, требовалось реализовать графическое представление автомата. Для теста №1 визуализация:



root – корень

Серые – терминальные вершины – вершины со словом

Черные ребра – добавленная буква

Красные пунктирные ребра – суффиксная ссылка

Рисунок 1 – Визуализация автомата, для входных данных №1

# Приложение А Исходный код программы

Название файла: Main.cpp

#include <iostream>

#include <vector>

#include <string>

#include <map>

#include <queue>

#include <algorithm>

#include <sstream>

#include <set>

#include <fstream>

#include <cstdlib> // для system()

using namespace std;

struct Node {

    map<char, Node\*> children;

    Node\* fail = nullptr;

    Node\* dict = nullptr;

    vector<pair<int, int>> pattern\_info; // pattern index and offset

    bool end = false;

    vector<int> output;

    string word;                  // Слово, заканчивающееся в этом узле (если есть)

    string id;                   // Уникальный идентификатор узла

    int patIndex = -1;        // Индекс шаблона (если узел является концом шаблона)

    Node(string id) : id(id) {}

    string toString() {

        ostringstream oss;

        oss << "Node{id='" << id << "', word='";

        if (!word.empty()) oss << word;

        oss << "', patternIndex=" << patIndex << "}";

        return oss.str();

    }

};

class AhoCorasick {

    Node\* root;

    vector<pair<string, int>> subpatterns; // subpattern and its offset

    int pattern\_len;

    vector<string> patterns;

    //дерево для стандартной работы алгоритма

    void buildTrie() {

        root = new Node("root");

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

            Node\* cur = root;

            string way;

            for (char c : patterns[i]) {

                way+=c;

                if (!cur->children[c]) {

                    cur->children[c] = new Node(way);

                }

                cur = cur->children[c];

            }

            cur->output.push\_back(i);

        }

    }

    //дерево для шаблона с джокером

    void SpecbuildTrie() {

        root = new Node("root");

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

            Node\* cur = root;

            string way;

            for (char c : subpatterns[i].first) {

                way+=c;

                if (!cur->children[c]) {

                    cur->children[c] = new Node(way);

                }

                cur = cur->children[c];

            }

            cur->end = true;

            cur->pattern\_info.emplace\_back(i, subpatterns[i].second);

        }

    }

    //создание суффиксных ссылок

    void buildFailureLinks() {

        queue<Node\*> q;

        root->fail = root;

        for (auto& pair : root->children) {

            pair.second->fail = root;

            q.push(pair.second);

        }

        while (!q.empty()) {

            Node\* cur = q.front();

            q.pop();

            for (auto& pair : cur->children) {

                Node\* child = pair.second;

                Node\* failNode = cur->fail;

                while (failNode != root && !failNode->children[pair.first]) {

                    failNode = failNode->fail;

                }

                child->fail = failNode->children[pair.first] ? failNode->children[pair.first] : root;

                child->output.insert(child->output.end(), child->fail->output.begin(), child->fail->output.end());

                q.push(child);

            }

        }

    }

public:

    //построение дерева для обычного случая

    void Aho(vector<string> pats){

        patterns = pats;

        buildTrie();

        buildFailureLinks();

    }

    //построение дерева для шаблона с джокером

    void SpecAho(const vector<pair<string, int>>& subpats, int len){

        subpatterns = subpats;

        pattern\_len = len;

        SpecbuildTrie();

        buildFailureLinks();

    }

    //поиск подстрок в строке

    vector<pair<int, int>> search(string text) {

        vector<pair<int, int>> results;

        Node\* cur = root;

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

            char c = text[i];

            while (cur != root && !cur->children[c]) {

                cur = cur->fail;

            }

            cur = cur->children[c] ? cur->children[c] : root;

            for (int patternIdx : cur->output) {

                results.emplace\_back(i - patterns[patternIdx].size() + 2, patternIdx + 1);

            }

        }

        return results;

    }

    //поиск вхождений подстроки с джокером

    vector<int> SpecSearch(const string& text) {

        vector<int> pattern\_pos(text.size() + 1, 0);

        Node\* cur = root;

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

            while (cur != root && !cur->children[text[i]]) {

                cur = cur->fail;

            }

            cur = cur->children[text[i]] ? cur->children[text[i]] : root;

            for (Node\* state = cur; state != root; state = state->fail) {

                if (state->end) {

                    for (auto& [pattern\_idx, offset] : state->pattern\_info) {

                        int index = i - subpatterns[pattern\_idx].first.size() + 1 - offset;

                        if (index >= 0 && index + pattern\_len <= text.size()) {

                            pattern\_pos[index]++;

                        }

                    }

                }

            }

        }

        vector<int> result;

        for (int i = 0; i <= text.size() - pattern\_len; ++i) {

            if (pattern\_pos[i] == subpatterns.size()) {

                result.push\_back(i + 1);

            }

        }

        return result;

    }

    //визуализация работы программы для стандартного случая

    void visualize() {

        cout << "=== Визуализация автомата Ахо-Корасик ===" << endl;

        queue<Node\*> q;

        set<Node\*> visited;

        q.push(root);

        visited.insert(root);

        while (!q.empty()) {

            Node\* current = q.front();

            q.pop();

            cout << "Узел " << current->id << ":";

            if (!current->word.empty()) cout << " (слово: " << current->word << ")";

            cout << endl;

            // Переходы

            if(!current->children.empty()){

                for (auto& entry : current->children) {

                    if(entry.second){

                    cout << "  --" << entry.first << "--> " << entry.second->id << endl;

                if (visited.find(entry.second) == visited.end()) {

                    visited.insert(entry.second);

                    q.push(entry.second);

                }

                    }

            }

        }

            if (current->fail!= nullptr && current->fail != root) {

                cout << "  --failure--> " << current->fail->id << endl;

            }

    }

        cout << "=== Конец визуализации ===" << endl;

    }

    //визуализация работы программы для случая с джокером

    void SpecVisualize(const string& pattern, char spec) {

        cout << "=== Визуализация поиска с джокером ===" << endl;

        cout << "Шаблон: " << pattern << endl;

        cout << "Джокер: '" <<spec << "'" << endl;

        cout << "Граф переходов:" << endl;

        cout << "[start]";

        for (char c : pattern) {

            cout << " --" << (c == spec ? "любой символ" : string(1, c)) << "--> ";

        }

        cout << "[end]" << endl;

        cout << "=== Конец визуализации ===" << endl;

    }

    //генерация dot файла дерева

        void generateDotFile(const string& filename) {

        ofstream dotFile(filename);

        dotFile << "digraph AhoCorasick {" << endl;

        dotFile << "    rankdir=LR;" << endl;

        dotFile << "    node [shape=circle];" << endl;

        queue<Node\*> q;

        set<Node\*> visited;

        q.push(root);

        visited.insert(root);

        // Сначала добавляем все узлы

        while (!q.empty()) {

            Node\* current = q.front();

            q.pop();

            // Определяем стиль узла

            string nodeStyle = "regular";

            if (current == root) {

                nodeStyle = "bold";

            }

            if (!current->output.empty()) {

                nodeStyle = "filled";

            }

            // Добавляем узел

            dotFile << "    \"" << current->id << "\" [style=" << nodeStyle;

            if (!current->output.empty()) {

                dotFile << ", fillcolor=lightgray";

            }

            dotFile << "];" << endl;

            // Добавляем переходы

            for (auto& entry : current->children) {

                if (entry.second) {

                    dotFile << "    \"" << current->id << "\" -> \"" << entry.second->id

                             << "\" [label=\"" << entry.first << "\"];" << endl;

                    if (visited.find(entry.second) == visited.end()) {

                        visited.insert(entry.second);

                        q.push(entry.second);

                    }

                }

            }

            // Добавляем fail-ссылки

            if (current->fail && current->fail != root) {

                dotFile << "    \"" << current->id << "\" -> \"" << current->fail->id

                         << "\" [style=dotted, color=red];" << endl;

            }

        }

        dotFile << "}" << endl;

        dotFile.close();

        cout << "Граф автомата сохранен в файл " << filename << endl;

        cout << "Для визуализации выполните команду: dot -Tpng " << filename << " -o automaton.png" << endl;

    }

//генерация изображения построенного дерева шаблонов

void generateAndShowGraph() {

    const std::string dotFile = "automaton.dot";

    const std::string pngFile = "automaton.png";

    // 1. Генерируем DOT-файл

    generateDotFile(dotFile);

    // 2. Конвертируем в PNG

    std::string command = "dot -Tpng " + dotFile + " -o " + pngFile;

    if (system(command.c\_str())) {

        std::cerr << "Ошибка: Graphviz не установлен или команда dot не работает." << std::endl;

        return;

    }

    // 3. Открываем PNG (кросс-платформенно)

    #ifdef \_WIN32

        command = "start " + pngFile;

    #elif \_\_APPLE\_\_

        command = "open " + pngFile;

    #else // Linux

        command = "xdg-open " + pngFile;

    #endif

    system(command.c\_str());

}

};

//функция для разбиени строки с джокером на подстроки

vector<pair<string, int>> split\_pattern(const string& pattern, char wild) {

    vector<pair<string, int>> subpatterns;

    string current;

    int start\_pos = 0;

    for (size\_t i = 0; i < pattern.size(); ++i) {

        if (pattern[i] != wild) {

            if (current.empty()) {

                start\_pos = i;

            }

            current += pattern[i];

        }

        else {

            if (!current.empty()) {

                subpatterns.emplace\_back(current, start\_pos);

                current.clear();

            }

        }

    }

    if (!current.empty()) {

        subpatterns.emplace\_back(current, start\_pos);

    }

    return subpatterns;

}

int main() {

    string text, pattern;

    getline(cin, text);          // Чтение текста

    getline(cin, pattern);    // Чтение второй строки

    bool isVisualize = true;     // Флаг для включения/отключения визуализации

    try {

        int n = stoi(pattern); // Попытка преобразовать вторую строку в число

        vector<string> patterns(n);

        for (int i = 0; i < n; i++) {

            getline(cin, patterns[i]); // Чтение шаблонов

        }

        AhoCorasick ac;

        ac.Aho(patterns);

        if (isVisualize) {

            ac.visualize();

            ac.generateAndShowGraph();       // Визуализация автомата

        }

        vector<pair<int, int>> result = ac.search(text);

        sort(result.begin(), result.end());

        for (auto& pair : result) { // Поиск и вывод результатов

            cout << pair.first<<" "<<pair.second << endl;

        }

    } catch (const invalid\_argument&) {

        // Если вторая строка не является числом, интерпретируем ее как шаблон с джокером

        char spec;

        string specStr;

        getline(cin, specStr);

        spec = specStr[0]; // Чтение символа джокера

        auto subpatterns = split\_pattern(pattern, spec);

        if (subpatterns.empty()) {

            cout << -1 << endl;

            return 0;

        }

        AhoCorasick ac;

        ac.SpecAho(subpatterns, pattern.size());

        if (isVisualize) {

            ac.SpecVisualize(pattern, spec);

            ac.generateAndShowGraph();   // Визуализация поиска с джокером

        }

        for (auto& pair : ac.SpecSearch(text)) { // Поиск и вывод результатов

            cout << pair<< endl;

        }

        }

    return 0;

}