**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 |  | Алтухов А.Д. |
| Преподаватель |  | Размочаева Н. В. |

Санкт-Петербург

2020

## Цель работы.

Построить алгоритм Ахо-Корасик для определения всех вхождений подстрок в строку, определить его сложность.

**Вариант 3.**

Вычислить длину самой длинной цепочки из суффиксных ссылок и самой длинной цепочки из конечных ссылок в автомате.

## Основные теоретические положения.

Первая строка содержит текст. Вторая – число n, далее – n строк.

Выход: все вхождения n в текст. 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.

## Описание алгоритма.

После считывания всех исходных данных стандартным образом строится бор. Далее в бор добавляются суффиксные ссылки с помощью поиска в ширину.

Поиск вхождений происходит следующим образом: обрабатывается текущий символ строки. Если в автомате существует ребро с этим символом, то осуществляется переход по нему. Если нет, то производится переход по суффиксной ссылке и проверка наличия ребра повторяется. Если встречается конечная вершина, то результат запоминается.

Так как таблица переходов автомата хранится в std::map, который реализован с помощью красно-черного дерева, временная сложность алгоритма составляет O((H + n) logq + k), расход памяти: O(n), где H – длина текста для поиска, n – длина всех подстрок, q – размер алфавита, k – общая длина всех совпадений.

Требуемая память: *O*(*n*), хранится только словарь.

Задание по поиску с джокерами реализуется построением бора по частям подстроки для поиска, разделенной джокерами, и дальнейшим объединением результатов.

Для индивидуального задания заведены функции deepestSuffix и deepestEnd, которые ищут самые длинные суффиксные и конечные ссылки. Для этого для вершин построенного бора запускаются рекурсивные переходы по суффиксным ссылкам. Самый большой возвращенный результат и есть ответ.

## Описание основных структур данных и функций.

class Trie – реализация бора.

std::vector<int> step(char direction) – шаг автомата.

int deepestSuffix(int vertex) – индивидуальное задание, нахождение самой длинной суффиксной ссылки.

int deepestEnd(int vertex) – индивидуальное задание, нахождение самой длинной конечной ссылки.

void search(std::string& str, std::vector<std::string>& words) – сборка автомата и запуск поиска.

**Тестирование.**

Таблица 1 – Результаты тестирования.

|  |  |
| --- | --- |
| **Ввод** | **Вывод** |
| NTAG  3  TAGT  TAG  T | 2 2  2 3 |
| ACTANCA  A$$A$  $ | 1 |

# Вывод.

В ходе работы был построен алгоритм Ахо-Корасик для поиска вхождений нескольких подстрок.

**ПРИЛОЖЕНИЕ А.  
ИСХОДНЫЙ КОД**

#include <iostream>

#include <algorithm>

#include <string>

#include <vector>

#include <locale>

#include <fstream>

#include <map>

#include <queue>

std::ostream\* out;

std::istream\* in;

struct Vertex {

std::map<char, int> paths;

int parent = 0;

int suffix = 0;

bool endOfWord = false;

int whatWord = -1;

};

class Trie {

std::vector<Vertex> vertexes;

int curVertex = 0;

public:

Trie() {

vertexes.push\_back(Vertex());

}

void addWords(std::vector<std::string>& words) { //собирание бора

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

int current = 0;

for (int j = 0; j < words[i].size(); j++){

if (vertexes[current].paths.find(words[i][j]) == vertexes[current].paths.end()) {

vertexes.push\_back(Vertex());

vertexes[current].paths[words[i][j]] = vertexes.size() - 1;

vertexes[vertexes.size() - 1].parent = current;

current = vertexes.size() - 1;

}

else {

current = vertexes[current].paths.find(words[i][j])->second;

}

}

if (vertexes[current].whatWord == -1) {

vertexes[current].whatWord = i;

vertexes[current].endOfWord = true; //endOfWord - передается всем суффиксам, whatWord только у действительного конца

}

}

makeSuffixLinks();

\*out << "Глубина суффиксной ссылки " << deepestSuffix(0) << "\n";

\*out << "Глубина конечной ссылки " << deepestEnd(0) << "\n";

}

void makeSuffixLinks() { //через бфс

std::queue<std::pair<char, int>> opened;

opened.push(std::make\_pair('\0', 0)); //корень

while (!opened.empty()){

auto current = opened.front();

opened.pop();

for (auto link : vertexes[current.second].paths){

opened.push(link);

}

vertexes[current.second].suffix = getSuffixLink(current.second, current.first);

//конечная ссылка

if (vertexes[vertexes[current.second].suffix].endOfWord){

vertexes[current.second].endOfWord = true;

}

}

}

int getSuffixLink(int vertex, char name){

int current = vertexes[vertex].parent;

int curSuffix = vertexes[current].suffix;

while (current != 0){

auto findedNode = vertexes[curSuffix].paths.find(name);

if (findedNode != vertexes[curSuffix].paths.end()){

return findedNode->second;

}

else{

current = curSuffix;

curSuffix = vertexes[current].suffix;

}

}

return 0;

}

std::vector<int> step(char direction){

std::vector<int> entries;

auto findedNode = vertexes[curVertex].paths.find(direction);

while (findedNode == vertexes[curVertex].paths.end() && curVertex != 0){

curVertex = vertexes[curVertex].suffix;

\*out << "Нет пути, переход по суффиксной ссылке в " << curVertex << "\n";

findedNode = vertexes[curVertex].paths.find(direction);

}

if (findedNode != vertexes[curVertex].paths.end()){

curVertex = vertexes[curVertex].paths.find(direction)->second;

\*out << "По суффиксной ссылке найден путь дальше в " << curVertex << "\n";

}

int entriePos = curVertex;

while (vertexes[entriePos].endOfWord){

if (vertexes[entriePos].whatWord != -1){

\*out << "Конец слова в " << entriePos << "\n";

\*out << "Найдено слово " << vertexes[entriePos].whatWord << "\n";

entries.push\_back(vertexes[entriePos].whatWord);

}

entriePos = vertexes[entriePos].suffix;

}

return entries;

}

int deepestSuffix(int vertex){

int deep = 0;

if (vertex == 0){

for (int i = 1; i < vertexes.size(); i++){

int buf = deepestSuffix(i);

if (buf > deep) {

deep = buf;

}

}

}

else{

if (vertexes[vertex].suffix == 0){

return 0;

}

else{

return deepestSuffix(vertexes[vertex].suffix) + 1;

}

}

return deep;

}

int deepestEnd(int vertex) {//считаем суффиксы с пометкой endOfWord

int deep = 0;

if (vertex == 0) {

for (int i = 1; i < vertexes.size(); i++) {

int buf = deepestEnd(i);

if (buf > deep) {

deep = buf;

}

}

}

else {

if (vertexes[vertex].suffix == 0 || !vertexes[vertex].endOfWord) {

return 0;

}

else {

return deepestEnd(vertexes[vertex].suffix) + 1;

}

}

return deep;

}

};

class Joker {

int substrSize = 0;

int partsCount = 0;

Trie trie;

std::map<std::string, std::vector<int>> parts;

public:

void makeParts(std::string& pattern, char joker) {

std::string word = "";

for (unsigned offset = 0; offset < pattern.size(); offset++){

if (pattern[offset] != joker){

word += pattern[offset];

}

if (pattern[offset + 1] == joker || offset + 1 >= pattern.size()){

if (!word.empty()){

auto finded = parts.find(word);

if (finded == parts.end()){

\*out << "Новое разбиение " << word << "\n";

std::vector<int> offsets;

offsets.push\_back(offset - word.size() + 1);

parts.emplace(word, offsets);

partsCount += 1;

}

else{

\*out << "Новое разбиение " << word << "\n";

finded->second.push\_back(offset - word.size() + 1);

partsCount += 1;

}

word = "";

}

}

}

}

std::map<int, int> findPossibleEntries(std::string& str) {

std::map<int, int> possibleEntries;

std::vector<std::string> dict;

for (auto pair : parts){

dict.push\_back(pair.first);

}

trie.addWords(dict);

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

for (auto entrie : trie.step(str[i])){

for (auto offset : parts.find(dict[entrie])->second){

int entriePos = i - dict[entrie].size() - offset + 1;

if (entriePos >= 0 && entriePos + substrSize <= static\_cast<int>(str.size())){

auto possibility = possibleEntries.find(entriePos);

if (possibility != possibleEntries.end()){

possibility->second += 1;

}

else{

possibleEntries.emplace(entriePos, 1);

}

}

}

}

}

return possibleEntries;

}

std::vector<int> getEntries(std::string& str, std::string& substr, char joker) {

std::vector<int> entries;

substrSize = substr.size();

makeParts(substr, joker);

auto possibleEntries = findPossibleEntries(str);

for (auto entrie : possibleEntries)

{

if (entrie.second == partsCount)

{

\*out << "Вхождение в " << entrie.first << "\n";

entries.push\_back(entrie.first);

}

}

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

return entries;

}

};

void search(std::string& str, std::vector<std::string>& words) {

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

Trie trie;

trie.addWords(words);

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

auto entrie = trie.step(str[i]);

if (!entrie.empty()){

for (int j = 0; j < entrie.size(); j++){

entries.push\_back(std::make\_pair(i - words[entrie[j]].size() + 1, entrie[j]));

}

}

}

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

for (auto entrie : entries){

\*out << entrie.first + 1 << " " << entrie.second + 1 << "\n";

}

}

void searchWithJoker(std::string& str, std::string& pattern, char joker) {

Joker jokerSearch;

auto entries = jokerSearch.getEntries(str, pattern, joker);

for (auto entrie : entries)

{

\*out << entrie + 1 << "\n";

}

}

int main()

{

setlocale(LC\_ALL, "Russian");

int mode;

int inputMode, outputMode;

std::cout << "Режим работы (0 - поиск всех вхождений в строку, 1 - поиск с джокером): ";

std::cin >> mode;

std::cout << "Ввод из... (0 - из консоли, 1 - из файла): ";

std::cin >> inputMode;

std::cout << "Вывод из... (0 - из консоли, 1 - из файла): ";

std::cin >> outputMode;

std::ifstream inFile("input.txt");

std::ofstream outFile("output.txt");

in = inputMode == 0 ? &std::cin : &inFile;

out = outputMode == 0 ? &std::cout : &outFile;

if (mode == 0) {

std::string str;

\*in >> str;

int patternsCount = 0;

\*in >> patternsCount;

std::vector<std::string> words;

std::string buf;

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

\*in >> buf;

words.push\_back(buf);

}

search(str, words);

}

else {

std::string str;

\*in >> str;

std::string pattern;

\*in >> pattern;

char joker;

\*in >> joker;

searchWithJoker(str, pattern, joker);

}

inFile.close();

outFile.close();

return 0;

}