**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: **Алгоритм Ахо-Корасик**

Вариант 5

|  |  |  |
| --- | --- | --- |
| Студент гр. 8303 |  | Удод М.Н. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2020

**Цель работы.**

Изучение алгоритма Ахо-Корасик.

**Задание 1.**

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

**Задание 2.**

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 P)

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

**Индивидуализация.**

Вычислить максимальное количество дуг, исходящих из одной вершины в боре; вырезать из строки поиска все найденные образцы и вывести остаток строки поиска.

**Описание алгоритма 1.**

В программе используется алгоритм Ахо-Карасика. Он заключается в построении бора и дополнением его суффикс-ссылка и терминальными ссылками.

В реализации алгоритма, используемой в программе, суффикс-ссылки и терминальные ссылки вычисляются лениво, то есть только тогда, когда они понадобятся. После первого вычисления они сохраняются и при последующем обращении берется сохраненное значение.

В программе используется функция перехода из текущей вершины по символу. Если у текущей вершины есть наследник, в которого можно пройти по заданному символу, то возвращается этот наследник. Если его нет и текущая вершина - корень, то возвращается корень. Если неверно и это, то вычисляется суффикс-ссылка родителя и для нее вызывается функция перехода по символу.

Суффикс-ссылка вычисляется следующим образом: если вершина, для которой требуется вычислить ссылку – корень или наследник корня, то возвращается ссылка на корень, иначе вычисляется суффикс-ссылка родителя и для нее вызывается функция перехода по символу, с помощью которого из родителя текущей вершины можно попасть в текущую вершину.

Терминальная ссылка определяется через суффикс-ссылку текущей вершины. Если она указывает на терминальную вершину или корень, то она и является терминальной ссылкой. Иначе возвращается терминальная ссылка вершины, на которую указывает суффикс-ссылка.

Поиск подстроки в тексте происходит с помощью посимвольного чтения текста. В зависимость от символа, выполняются переходы из текущей вершины в вычисленную с помощью функции перехода. Если в результате переходов текущая вершина стала терминальной, то совпадение найдено. Так же после каждого переходы просматриваются вершины, на которые указывает терминальная ссылка. Если она является терминальной, то найдено еще одно совпадение. Переход по терминальным ссылкам происходит до тех пор, пока она не укажет на корень.

При построении бора происходит проход по всем символам всех подстрок. Временная сложность этого этапа O(P), где P – сумма всех длин подстрок. Так как переходы хранятся в std::map, реализованном с помощью красно-черных деревьев, то сложность вставки элемента O(log(A)), где A – количество символов в алфавите. Результирующая сложность построения бора: O(P\*log(A)).

Далее, для поиска подстроки происходит посимвольный проход по тексту. Это дает сложность O(L), где L – длина текста. Так как при проходе по бору требуется искать элемент в std::map, то получаем результирующую сложность O(L\*log(A)). Так же на сложность влияют переходы по терминальным ссылкам. Количество таких переходов равно количеству всех возможных вхождений всех подстрок в текст. Обозначим эту величину как t.

Результирующая временная сложность алгоритма: O((P+L)\*log(A)+t).

Для работы алгоритма требуется хранения бора. В программе для этого используется std::map, хранящий только нужные переходы. Отсюда сложность по памяти O(P).

**Описание алгоритма 2.**

Для поиска шаблона, содержащего джокер, строка разбивается на части по символу джокеру. Для каждой такой части сохраняется ее место в первоначальном шаблоне. Получившиеся подстроки ищутся в тексте с помощью алгоритма Ахо-Корасика. При каждом совпадении вычисляется место, которому соответствует текущая подстрока. Для каждого такого места хранится количество найденных в нем совпадений. Если в каком либо месте число совпадений достигнет количества подстрок, то найдено соответствие шаблону.

Так как в программе используется алгоритм Ахо-Корасик, то временная сложность O((P+L)\*log(A)+t), где P – длина шаблона, L – длина текста, A – количество символов в алфавите, t – количество совпадений подстрок.

Сложность по памяти алгоритма Ахо-Корасик – O(P). Так же требуется память для хранения числа совпадений для каждого места. Результирующая сложность по памяти – O(P+t).

**Описание структур данных.**

Класс BarNode:

int depth – расстояние от текущей вершины до корня  
BorNode \*parent – ссылка на родителя  
char parentThrough – символ, через который из родителя можно попасть в текущую вершину  
BorNode \*suffixPtr – суффикс-ссылка  
BorNode \*terminalPtr - терминальная ссылка  
bool \_isTerminal – является ли текущая вершина терминальной  
std::vector<int> \_needles – если вершина терминальная, то для каких подстрок  
std::map<char, BorNode\*> children - наследники

**Описание функций.**

std::vector<std::pair<std::string, int>> split(std::string strToSplit, char delimeter)

strToSplit – строка, которую требуется разделить

delimetr – символ, по которому требуется разделить строку

Функция разделяет строку по символу и возвращает массив пар, первый элемент которой – подстрока, а второй – индекс начала подстроки в исходной строке

Класс BarNode:

void addNeedle(std::string &needle, int needleCount)

needle – подстрока, которую нужно добавить в бор

needleCount – номер подстроки

Функция добавляет подстроку в бор

BorNode \*getSuffixPtr()

Функция возвращает суффикс-ссылку для текущей вершины

BorNode \*getTerminalPtr()

Функция возвращает терминальную ссылку для текущей вершины

BorNode \*deltaPtr(char c)

c – символ, по которому требуется перейти

Функция реализует функцию перехода и возвращает вершину, в которую следует перейти из текущей вершины по символу c.

void forEach(std::function<void(BorNode\*)> f)

f – функция, которую требуется применить к каждой вершин бора

Функция применяет функцию f к каждой вершине бора

**Тестирование алгоритма 1.**

CCCA

1

CC

1 1

2 1

Максимальное число исходящие ребер: 1

Текст с вырезанными подстроками: A

aaabaabba

1

b

4 1

7 1

8 1

Максимальное число исходящие ребер: 1

Текст с вырезанными подстроками: aaaaaa

abbbabbbabab

2

abbba

bab

1 1

4 2

5 1

8 2

10 2

Максимальное число исходящие ребер: 2

Текст с вырезанными подстроками:

GAGTCATTGAGTGTTC

7

G

AA

TTC

TT

TGATGT

TG

ATTCATCT

1 1

3 1

7 4

8 6

9 1

11 1

12 6

13 1

14 4

14 3

Максимальное число исходящие ребер: 3

Текст с вырезанными подстроками: ATCAA

ababababa

2

ab

ba

1 1

2 2

3 1

4 2

5 1

6 2

7 1

8 2

Максимальное число исходящие ребер: 2

Текст с вырезанными подстроками:

acccca

2

ac

cc

1 1

2 2

3 2

4 2

Максимальное число исходящие ребер: 2

Текст с вырезанными подстроками: a

Тестирование алгоритма 2.

ACT

A$

$

1

GTTGAAG

G$$G

$

1

4

AAAB

A@

@

1

2

3

AAB

AAB

$

1

GTAGADGGGAGG

CVCVCV

V

GATGAGCGAGAGAGCC

GA#

#

1

4

8

10

12

ACCC

A%%%

%

1

Вывод.

В ходе выполнения лабораторной работы был изучен алгоритм Ахо-Корасика для нахождения подстроки в строке.

Приложения А. Исходный код

Алгоритм 1.

#include <iostream>  
#include <map>  
#include <vector>  
#include <functional>  
  
//#define DEBUG  
  
#ifdef DEBUG  
  
int globalIdCounter = 0;  
  
#endif  
  
// Функция разделения строки на подшаблоны  
std::vector<std::pair<std::string, int>> split(std::string strToSplit, char delimeter)  
{  
 std::vector<std::pair<std::string, int>> splitted;  
  
 std::string nowStr = "";  
  
 for (int i=0; i<strToSplit.size(); i++){  
 // Если встречаем разделитель и это не несколько шаблонов подряд, то есть нашли новый подшаблон  
 if (strToSplit[i] == delimeter && !nowStr.empty()){  
 splitted.emplace\_back(nowStr, i-nowStr.size());  
 nowStr = "";  
 } else if(strToSplit[i] != delimeter){  
 nowStr += strToSplit[i]; // Иначе добавляем текущий символ в текущий подшаблон  
 }  
 }  
  
 // В результате выполнения цикла выше последний подшаблон мог быть не добавлен  
 if (!nowStr.empty()){  
 splitted.emplace\_back(nowStr, strToSplit.size()-nowStr.size());  
 }  
  
 return splitted;  
}  
  
class BorNode{  
  
private:  
 int depth;  
  
 BorNode \*parent = nullptr;  
 char parentThrough;  
 BorNode \*suffixPtr = nullptr;  
 BorNode \*terminalPtr = nullptr;  
  
 bool \_isTerminal = false;  
 std::vector<int> \_needles = {};  
  
 std::map<char, BorNode\*> children = {};  
  
  
public:  
  
 BorNode(): parentThrough(0), depth(0) {  
#ifdef DEBUG  
 id = globalIdCounter;  
 globalIdCounter++;  
#endif  
 }  
 explicit BorNode(BorNode \*parent, char through): parent(parent), parentThrough(through), depth(parent->depth+1) {  
#ifdef DEBUG  
 id = globalIdCounter;  
 globalIdCounter++;  
#endif  
 }  
  
 // Добавление шаблона в бор  
 void addNeedle(std::string &needle, int needleCount){  
  
#ifdef DEBUG  
 std::cout << std::endl;  
 std::cout << "\tТекущая вершина: " << id << std::endl;  
#endif  
  
 // Если есть необработанные символы  
 if (!needle.empty()) {  
  
#ifdef DEBUG  
 std::cout << "\tТекущий символ: " << needle[0] << std::endl;  
#endif  
  
 auto \*next = children[needle[0]]; // Элемент, в который можем перейти по текущему символу  
  
 // Если по этому пути еще не ходили  
 if (next == nullptr) {  
  
 next = new BorNode(this, needle[0]); // То создаем этот путь  
 children[needle[0]] = next;  
#ifdef DEBUG  
 std::cout << "\tНет наследника по этому символу. Создаю. Его ID: " << next->id << std::endl;  
#endif  
  
 } else{  
#ifdef DEBUG  
 std::cout << "\tНаследник обнаружен. Его ID: " << next->id << std::endl;  
#endif  
 }  
  
 // Переходим к следующему символу  
 needle.erase(0, 1);  
 next->addNeedle(needle, needleCount);  
  
 } else{  
  
#ifdef DEBUG  
 std::cout << "\tСтрока пустая. Значит, вершина - терминальная." << std::endl;  
#endif  
  
 // Так как дошли до конца строки, то отмечаем вершину терминальной  
 \_isTerminal = true;  
 \_needles.push\_back(needleCount);  
  
 }  
  
 }  
  
 // Получение суффикс-ссылки. Реализована лениво.  
 BorNode \*getSuffixPtr(){  
  
#ifdef DEBUG  
 std::cout << std::endl;  
 std::cout << "\tРассчет суфикс-ссылки. ID вершины: " << id << std::endl;  
#endif  
  
 // Если текущая верщина - корень, то возвращаем его в качестве суффикса  
 if (parent == nullptr){  
#ifdef DEBUG  
  
 std::cout << "\tТекущая вершина - корень" << std::endl;  
  
#endif  
 return this;  
 }  
  
 // Если на уровень ниже корня, то возвращаем корень  
 if (parent->parent == nullptr){  
#ifdef DEBUG  
  
 std::cout << "\tТекущая вершина - сын корня." << std::endl;  
  
#endif  
 return parent;  
 }  
  
 // Если суффикс-ссылка еще не вычислена, то вычисляем  
 if (suffixPtr == nullptr){  
#ifdef DEBUG  
 std::cout << "\tСуффикс ссылка еще не вычислялась. Переход к суффикс-ссылке родителя. ID родителя: " << parent->id << std::endl;  
#endif  
  
 // Переходим по суффикс-ссылке родителя и смотрим, можно ли из нее пройти по символу, по которому попали сюда  
 suffixPtr = parent->getSuffixPtr()->deltaPtr(parentThrough);  
 }else{  
#ifdef DEBUG  
  
 std::cout << "\tСуффикс-ссылка уже вычислялась." << std::endl;  
  
#endif  
 }  
  
#ifdef DEBUG  
  
 std::cout << "\tСуффикс-ссылка вычеслена." << std::endl;  
  
#endif  
  
 return suffixPtr;  
  
 };  
  
 // Вычисление терминальной ссылке. Реализована лениво.  
 BorNode \*getTerminalPtr(){  
  
#ifdef DEBUG  
 std::cout << std::endl;  
 std::cout << "\tРасчет терминальной ссылки. ID вершины: " << id << std::endl;  
  
#endif  
  
 // Если корень, то возврашаем его  
 if (isRoot()){  
#ifdef DEBUG  
  
 std::cout << "\tТекущая вершина - корень" << std::endl;  
  
#endif  
 return this;  
 }  
  
 // Если еще не вычисляли  
 if (terminalPtr == nullptr){  
  
#ifdef DEBUG  
 std::cout << "\tТерминальная ссылка еще не вычислялась." << std::endl;  
#endif  
  
 if (getSuffixPtr()->\_isTerminal){  
#ifdef DEBUG  
 std::cout << "\tВершина по суффикс-ссылке терминальная." << std::endl;  
#endif  
 terminalPtr = getSuffixPtr();  
 } else if (getSuffixPtr()->isRoot()){  
#ifdef DEBUG  
 std::cout << "\tВершина по суффикс-ссылке - корень." << std::endl;  
#endif  
 terminalPtr = getSuffixPtr();  
 } else{  
#ifdef DEBUG  
 std::cout << "\tПереход по терминальной ссылке суффикс-ссылки."<< std::endl;  
#endif  
 terminalPtr = getSuffixPtr()->getTerminalPtr();  
 }  
  
 } else{  
#ifdef DEBUG  
 std::cout << "\tТерминальная ссылка уже вычислялась." << std::endl;  
#endif  
 }  
  
 return terminalPtr;  
  
 }  
  
 // Вычисление ссылки, в которую нужно перейти по переданному символу.  
 // Данная функция учитывает, что если прямого перехода по символу нет, то требуется перейти по суффикс ссылке  
 BorNode \*deltaPtr(char c){  
  
#ifdef DEBUG  
  
 std::cout << "\tРассчет перехода по символу: " << c << std::endl;  
 std::cout << "\tID текущей вершины: " << id << std::endl;  
  
#endif  
  
 // Проверяем возможность прямого перехода  
 BorNode \*child = children[c];  
  
 if (child == nullptr){  
#ifdef DEBUG  
  
 std::cout << "\tНет сына по этому символу" << std::endl;  
  
#endif  
 if (isRoot()){  
#ifdef DEBUG  
  
 std::cout << "\tПопали в корень. Так как из него нет вершины, то остаемся тут." << std::endl;  
  
#endif  
 return this;  
 }  
  
#ifdef DEBUG  
  
 std::cout << "\tИспользуем суффикс-ссылку в качестве промежуточной вершины." << std::endl;  
  
#endif  
 // Если нет прямого перехода, то переходим по суффикс-ссылке  
 return getSuffixPtr()->deltaPtr(c);  
  
 }  
  
#ifdef DEBUG  
  
 std::cout << "\tОбнаружен прямой путь. ID вершины: " << child->id << std::endl;  
  
#endif  
  
 return child;  
  
 }  
  
 // Является ли текущая вершина - терминальной  
 bool isTerminal(){ return \_isTerminal; }  
 // Является ли текущая вершина еорнем  
 bool isRoot(){ return parent == nullptr; }  
  
 int getDepth(){ return depth; }  
 char getParentThrough(){ return parentThrough; }  
 std::map<char, BorNode\*> &getChildren(){ return children; }  
 std::vector<int> &getNeedles(){ return \_needles; }  
  
 void forEach(std::function<void(BorNode\*)> f){  
  
 f(this);  
 for (auto child: children){  
 if (child.second != nullptr)  
 child.second->forEach(f);  
 }  
  
 }  
  
#ifdef DEBUG  
 int id;  
#endif  
};  
  
int main() {  
  
 std::string text;  
 std::string needle;  
 // Найденные совпадения в формате <индекс начала, индекс конца>  
 std::vector<std::pair<int, int>> ans;  
  
 auto \*root = new BorNode();  
  
 std::cin >> text;  
  
 int needleCount;  
 std::cin >> needleCount;  
  
#ifdef DEBUG  
 std::cout << "Построение бора: " << std::endl;  
#endif  
 for (int needleIndex=0; needleIndex<needleCount; needleIndex++){  
 std::cin >> needle;  
#ifdef DEBUG  
 std::cout << "Добавление строки " << needle << std::endl;  
#endif  
 root->addNeedle(needle, needleIndex);  
 }  
  
#ifdef DEBUG  
 std::cout << "Вывод построенного бора: " << std::endl;  
 root->forEach([](BorNode \*now){  
 std::cout << std::endl;  
 std::cout << "Вершина с ID: " << now->id << std::endl;  
 std::cout << "\tТерминальная: " << now->isTerminal() << std::endl;  
 std::cout << "\tМожно попасть в: " << std::endl;  
 for (auto child: now->getChildren()){  
 std::cout << "\t\t" << "В вершину с ID " << child.second->id << " через символ " << child.first << std::endl;  
 }  
  
 });  
#endif  
  
#ifdef DEBUG  
  
 std::cout << "Поиск шаблона в тексте: " << std::endl;  
  
#endif  
  
 BorNode \*now = root;  
 // Проходим по бору в зависимости от приходящих символов  
 for (int i=0; i<text.size(); i++){  
  
 char c = text[i];  
  
#ifdef DEBUG  
 std::cout << std::endl;  
 std::cout << "Текущий символ: " << c << std::endl;  
  
#endif  
  
 // Получаем вершину, в которую надо перейти  
 now = now->deltaPtr(c);  
  
#ifdef DEBUG  
  
 std::cout << "Текущая вершина: " << now->id << std::endl;  
  
#endif  
  
 // Если она терминальня, то нашли одно из совпадений  
 if (now->isTerminal()){  
  
 for (auto needleIndex: now->getNeedles()) {  
#ifdef DEBUG  
 std::cout << "Обнаружено совпадение: ";  
#endif  
 std::cout << i - now->getDepth() + 2 << " " << needleIndex + 1 << std::endl;  
 ans.emplace\_back(i - now->getDepth() + 2, i);  
 }  
  
#ifdef DEBUG  
  
 std::cout << "Попали в терминальную вершину. Начинаем обход терминальных ссылок." << std::endl;  
  
#endif  
  
 }  
  
 // Просмотр терминальных ссылок  
 BorNode \*terminalNext = now->getTerminalPtr();  
 // До тех пор, пока не попадем в корень  
 while (!terminalNext->isRoot()){  
  
#ifdef DEBUG  
  
 std::cout << "Найдено подслово по терминальной ссылке. ID терминальной вершины: " << terminalNext->id << std::endl;  
  
#endif  
 for (auto needleIndex: terminalNext->getNeedles()) {  
#ifdef DEBUG  
 std::cout << "Обнаружено совпадение: ";  
#endif  
 std::cout << i - terminalNext->getDepth() + 2 << " " << needleIndex + 1 << std::endl;  
 ans.emplace\_back(i - terminalNext->getDepth() + 2, i);  
 }  
 terminalNext = terminalNext->getTerminalPtr();  
  
 }  
  
 }  
  
 // Поиск максимального числа выходящих ребер бора  
 int maxOut = 0;  
  
 root->forEach([&maxOut](BorNode \*now){  
  
 int nowOut = 0;  
 for (auto child: now->getChildren()){  
 if (child.second != nullptr) nowOut++;  
 }  
 if (maxOut < nowOut) maxOut = nowOut;  
  
 });  
  
 std::cout << "Максимальное число исходящие ребер: " << maxOut << std::endl;  
  
 // Удаление вхождений. Так как могут быть пересечения, то отмечаем все символы, который должны быть удалены  
 for (auto founded: ans){  
 for (int i=founded.first-1; i<=founded.second; i++){  
 text[i] = '#'; // Это символа нет в алфавите  
 }  
 }  
  
 std::string erasedText;  
 for (auto c: text){  
  
 if (c != '#')  
 erasedText += c;  
  
 }  
  
 std::cout << "Текст с вырезанными подстроками: " << erasedText << std::endl;  
  
  
 return 0;  
}

Алгоритм 2.

#include <iostream>

#include <map>

#include <vector>

#include <functional>

#define DEBUG

#ifdef DEBUG

int globalIdCounter = 0;

#endif

// Функция разделения строки на подшаблоны

std::vector<std::pair<std::string, int>> split(std::string strToSplit, char delimeter)

{

std::vector<std::pair<std::string, int>> splitted;

std::string nowStr = "";

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

// Если встречаем разделитель и это не несколько шаблонов подряд, то есть нашли новый подшаблон

if (strToSplit[i] == delimeter && !nowStr.empty()){

splitted.emplace\_back(nowStr, i-nowStr.size());

nowStr = "";

} else if(strToSplit[i] != delimeter){

nowStr += strToSplit[i]; // Иначе добавляем текущий символ в текущий подшаблон

}

}

// В результате выполнения цикла выше последний подшаблон мог быть не добавлен

if (!nowStr.empty()){

splitted.emplace\_back(nowStr, strToSplit.size()-nowStr.size());

}

return splitted;

}

class BorNode{

private:

int depth;

BorNode \*parent = nullptr;

char parentThrough;

BorNode \*suffixPtr = nullptr;

BorNode \*terminalPtr = nullptr;

bool \_isTerminal = false;

std::vector<int> \_needles = {};

std::map<char, BorNode\*> children = {};

public:

BorNode(): parentThrough(0), depth(0) {

#ifdef DEBUG

id = globalIdCounter;

globalIdCounter++;

#endif

}

explicit BorNode(BorNode \*parent, char through): parent(parent), parentThrough(through), depth(parent->depth+1) {

#ifdef DEBUG

id = globalIdCounter;

globalIdCounter++;

#endif

}

// Добавление шаблона в бор

void addNeedle(std::string &needle, int needleCount){

#ifdef DEBUG

std::cout << std::endl;

std::cout << "\tТекущая вершина: " << id << std::endl;

#endif

// Если есть необработанные символы

if (!needle.empty()) {

#ifdef DEBUG

std::cout << "\tТекущий символ: " << needle[0] << std::endl;

#endif

auto \*next = children[needle[0]]; // Элемент, в который можем перейти по текущему символу

// Если по этому пути еще не ходили

if (next == nullptr) {

next = new BorNode(this, needle[0]); // То создаем этот путь

children[needle[0]] = next;

#ifdef DEBUG

std::cout << "\tНет наследника по этому символу. Создаю. Его ID: " << next->id << std::endl;

#endif

} else{

#ifdef DEBUG

std::cout << "\tНаследник обнаружен. Его ID: " << next->id << std::endl;

#endif

}

// Переходим к следующему символу

needle.erase(0, 1);

next->addNeedle(needle, needleCount);

} else{

#ifdef DEBUG

std::cout << "\tСтрока пустая. Значит, вершина - терминальная." << std::endl;

#endif

// Так как дошли до конца строки, то отмечаем вершину терминальной

\_isTerminal = true;

\_needles.push\_back(needleCount);

}

}

// Получение суффикс-ссылки. Реализована лениво.

BorNode \*getSuffixPtr(){

#ifdef DEBUG

std::cout << std::endl;

std::cout << "\tРассчет суфикс-ссылки. ID вершины: " << id << std::endl;

#endif

// Если текущая верщина - корень, то возвращаем его в качестве суффикса

if (parent == nullptr){

#ifdef DEBUG

std::cout << "\tТекущая вершина - корень" << std::endl;

#endif

return this;

}

// Если на уровень ниже корня, то возвращаем корень

if (parent->parent == nullptr){

#ifdef DEBUG

std::cout << "\tТекущая вершина - сын корня." << std::endl;

#endif

return parent;

}

// Если суффикс-ссылка еще не вычислена, то вычисляем

if (suffixPtr == nullptr){

#ifdef DEBUG

std::cout << "\tСуффикс ссылка еще не вычислялась. Переход к суффикс-ссылке родителя. ID родителя: " << parent->id << std::endl;

#endif

// Переходим по суффикс-ссылке родителя и смотрим, можно ли из нее пройти по символу, по которому попали сюда

suffixPtr = parent->getSuffixPtr()->deltaPtr(parentThrough);

}else{

#ifdef DEBUG

std::cout << "\tСуффикс-ссылка уже вычислялась." << std::endl;

#endif

}

#ifdef DEBUG

std::cout << "\tСуффикс-ссылка вычеслена." << std::endl;

#endif

return suffixPtr;

};

// Вычисление терминальной ссылке. Реализована лениво.

BorNode \*getTerminalPtr(){

#ifdef DEBUG

std::cout << std::endl;

std::cout << "\tРасчет терминальной ссылки. ID вершины: " << id << std::endl;

#endif

// Если корень, то возврашаем его

if (isRoot()){

#ifdef DEBUG

std::cout << "\tТекущая вершина - корень" << std::endl;

#endif

return this;

}

// Если еще не вычисляли

if (terminalPtr == nullptr){

#ifdef DEBUG

std::cout << "\tТерминальная ссылка еще не вычислялась." << std::endl;

#endif

if (getSuffixPtr()->\_isTerminal){

#ifdef DEBUG

std::cout << "\tВершина по суффикс-ссылке терминальная." << std::endl;

#endif

terminalPtr = getSuffixPtr();

} else if (getSuffixPtr()->isRoot()){

#ifdef DEBUG

std::cout << "\tВершина по суффикс-ссылке - корень." << std::endl;

#endif

terminalPtr = getSuffixPtr();

} else{

#ifdef DEBUG

std::cout << "\tПереход по терминальной ссылке суффикс-ссылки."<< std::endl;

#endif

terminalPtr = getSuffixPtr()->getTerminalPtr();

}

} else{

#ifdef DEBUG

std::cout << "\tТерминальная ссылка уже вычислялась." << std::endl;

#endif

}

return terminalPtr;

}

// Вычисление ссылки, в которую нужно перейти по переданному символу.

// Данная функция учитывает, что если прямого перехода по символу нет, то требуется перейти по суффикс ссылке

BorNode \*deltaPtr(char c){

#ifdef DEBUG

std::cout << "\tРассчет перехода по символу: " << c << std::endl;

std::cout << "\tID текущей вершины: " << id << std::endl;

#endif

// Проверяем возможность прямого перехода

BorNode \*child = children[c];

if (child == nullptr){

#ifdef DEBUG

std::cout << "\tНет сына по этому символу" << std::endl;

#endif

if (isRoot()){

#ifdef DEBUG

std::cout << "\tПопали в корень. Так как из него нет вершины, то остаемся тут." << std::endl;

#endif

return this;

}

#ifdef DEBUG

std::cout << "\tИспользуем суффикс-ссылку в качестве промежуточной вершины." << std::endl;

#endif

// Если нет прямого перехода, то переходим по суффикс-ссылке

return getSuffixPtr()->deltaPtr(c);

}

#ifdef DEBUG

std::cout << "\tОбнаружен прямой путь. ID вершины: " << child->id << std::endl;

#endif

return child;

}

// Является ли текущая вершина - терминальной

bool isTerminal(){ return \_isTerminal; }

// Является ли текущая вершина еорнем

bool isRoot(){ return parent == nullptr; }

int getDepth(){ return depth; }

char getParentThrough(){ return parentThrough; }

std::map<char, BorNode\*> &getChildren(){ return children; }

std::vector<int> &getNeedles(){ return \_needles; }

void forEach(std::function<void(BorNode\*)> f){

f(this);

for (auto child: children){

if (child.second != nullptr)

child.second->forEach(f);

}

}

#ifdef DEBUG

int id;

#endif

};

int main() {

std::string text;

std::string needle;

char joker;

auto \*root = new BorNode();

std::cin >> text;

std::cin >> needle;

std::cin >> joker;

auto subTemplatesInfo = split(needle, joker);

#ifdef DEBUG

std::cout << "Построение бора: " << std::endl;

#endif

for (int needleIndex=0; needleIndex<subTemplatesInfo.size(); needleIndex++){

#ifdef DEBUG

std::cout << "Добавление строки " << subTemplatesInfo[needleIndex].first << std::endl;

#endif

root->addNeedle(subTemplatesInfo[needleIndex].first, needleIndex);

}

#ifdef DEBUG

std::cout << "Вывод построенного бора: " << std::endl;

root->forEach([](BorNode \*now){

std::cout << std::endl;

std::cout << "Вершина с ID: " << now->id << std::endl;

std::cout << "\tТерминальная: " << now->isTerminal() << std::endl;

std::cout << "\tМожно попасть в: " << std::endl;

for (auto child: now->getChildren()){

std::cout << "\t\t" << "В вершину с ID " << child.second->id << " через символ " << child.first << std::endl;

}

});

#endif

#ifdef DEBUG

std::cout << "Поиск шаблона в тексте: " << std::endl;

#endif

std::map<int, int> possibleVars;

BorNode \*now = root;

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

char c = text[i];

#ifdef DEBUG

std::cout << std::endl;

std::cout << "Текущий символ: " << c << std::endl;

#endif

now = now->deltaPtr(c);

#ifdef DEBUG

std::cout << "Текущая вершина: " << now->id << std::endl;

#endif

if (now->isTerminal()){

for (auto needleIndex: now->getNeedles()) {

int possibleIndex = i - subTemplatesInfo[needleIndex].second - now->getDepth() + 1;

#ifdef DEBUG

std::cout << "Обнаружен подшаблон в позиции " << possibleIndex << std::endl;

#endif

possibleVars[possibleIndex] += 1;

if (possibleVars[possibleIndex] == subTemplatesInfo.size()) {

#ifdef DEBUG

std::cout << "Обнаружено совпадение шаблону в позиции: " << std::endl;

#endif

std::cout << possibleIndex+1 << std::endl;

}

}

}

BorNode \*terminalNext = now->getTerminalPtr();

while (!terminalNext->isRoot()){

#ifdef DEBUG

std::cout << "Найдено подслово по терминальной ссылке" << std::endl;

#endif

for (auto needleIndex: terminalNext->getNeedles()) {

int possibleIndex = i - subTemplatesInfo[needleIndex].second - terminalNext->getDepth() + 1;

#ifdef DEBUG

std::cout << "Обнаружен подшаблон в позиции " << possibleIndex << std::endl;

#endif

possibleVars[possibleIndex] += 1;

if (possibleVars[possibleIndex] == subTemplatesInfo.size()) {

#ifdef DEBUG

std::cout << "Обнаружено совпадение шаблону в позиции: " << std::endl;

#endif

std::cout << possibleIndex+1 << std::endl;

}

}

terminalNext = terminalNext->getTerminalPtr();

}

}

return 0;

}