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

Студен гр. 1381 _____ Возмитель В. Е.

Преподаватель _____ Токарев А. П

Санкт-Петербург

2023

Цель работы.

Изучить алгоритм Ахо-Корасик и применить его на практике.

Задание.

1. Разработайте программу, решающую задачу точного поиска набора образцов.

2. 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 необходимо найти все вхождения P в текст T.

Например, образец ab??c? 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 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}

Ход работы и описание алгоритма.

Входные данные: текст(txt) и массив подстрок(pttrns).

1) По массиву pttrns строится бор.

2) Выполняется цикл по тексту.

2.1) Поиск текущего символа в боре.

2.2) Если символ присутствует в боре и этот символ является окончанием какой-либо подстроки - подстрока найдена.

Для второго задания алгоритм - следующий:

1) Каждая подстрока массива pttrns разбивается по символу джокера. Пример: подстрока ‘aba$$asd’ с джокером ‘$’ будет разбита на две подстроки ‘aba’ и ‘asd’.

2) С помощью алгоритма из Задания 1 выполняется поиск подстрок, полученных в пункте 1.

3) Строится массив C. C[i] = количество встретившихся в тексте безмасочных подстрок шаблона, который начинается в тексте на позиции i.

4) Если C[i] равно длине подстроки – подстрока найдена.

Вычислительная сложность алгоритма в первом и втором задании равна O(a + h + k), где a — суммарная длина подстрок, h — длина текста, k — общая длина всех совпадений.

Описание основных функций и переменных:

Переменные:

txt— исходный текст.

pttrns — массив подстрок.

n — количество подстрок.

Tree — бор.

Классы:

Node — описание узла дерева.

У класса Node есть следующие поля:

child — словарь дочерних узлов.

suffix_link — суффиксная ссылка на узел.

words — массив слов на которых заканчивается узел.

Функции:

create_tree(pttrns) — по подстрокам создает дерево.

create_links(tree) — добавляет в дерево tree суффиксные ссылки.

Aho(string, pttrns) — поиск подстрок в тексте.

Выводы.

Был изучен алгоритм Ахо-Корасик, а также была произведена реализация данного алгоритма на языке программирования *python*.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла PIAA_5_1.py:

```
class Node:
    def __init__(self, link=None):
        self.child = {}
        self.suffix_link = link
        self.words = []


def create_tree(pttrns):
    root = Node()

    for index, pattern in enumerate(pttrns):
        node = root
        for c in pattern:
            node = node.child.setdefault(c, Node(root))

        node.words.append(index)

    return root


def create_links(tree):
    queue = [value for key, value in tree.child.items()]

    while queue:
        current_node = queue.pop(0)

        for key, value_node in current_node.child.items():
            queue.append(value_node)
            link_node = current_node.suffix_link

            while link_node is not None and key not in link_node.child:
                link_node = link_node.suffix_link

            value_node.suffix_link = link_node.child[key] if link_node
else tree
            value_node.words += value_node.suffix_link.words

    return tree


def Aho(string, ptrns):
    tree_root = create_links(create_tree(ptrns))
    res = []
    node = tree_root

    for i in range(len(string)):
        while node is not None and string[i] not in node.child:
            node = node.suffix_link

        if node is None:
```

```
            node = tree_root
            continue

        node = node.child[string[i]]

        for pttrn in node.words:
            res.append((i - len(ptrns[pttrn]) + 2, pttrn + 1))

    return res


txt = input()
n = int(input())
ptrns = []

for i in range(n):
    ptrns.append(input())

res = sorted(Aho(txt, ptrns))
for i in res:
    print(i[0], i[1])
```

Название файла PIAA_5_2.py:

```
class Node:
    def __init__(self, link=None):
        self.child = {}
        self.suffix_link = link
        self.words = []


def create_tree(pttrns):
    root = Node()

    for index, pattern in enumerate(pttrns):
        node = root
        for c in pattern:
            node = node.child.setdefault(c, Node(root))

        node.words.append(index)

    return root


def create_links(tree):
    queue = [value for key, value in tree.child.items()]

    while queue:
        current_node = queue.pop(0)

        for key, value_node in current_node.child.items():
            queue.append(value_node)
            link_node = current_node.suffix_link
```

```
            while link_node is not None and key not in link_node.child:
                link_node = link_node.suffix_link

            value_node.suffix_link = link_node.child[key] if link_node
else tree
            value_node.words += value_node.suffix_link.words

    return tree


def Aho(string, pttrns):
    tree_root = create_links(create_tree(pttrns))
    res = []
    node = tree_root

    for i in range(len(string)):
        while node is not None and string[i] not in node.child:
            node = node.suffix_link

        if node is None:
            node = tree_root
            continue

        node = node.child[string[i]]

        for pattern in node.words:
            res.append((i - len(pttrns[pattern]) + 2, pattern + 1))

    return res


def create_patterns(pttrn, wild_card):
    ptrns = list(filter(None, pttrn.split(wild_card)))
    start_indices = []
    start_index = 0

    for p in ptrns:
        tmp = pttrn[start_index:]
        index = tmp.index(p)
        start_indices.append(index + (len(pttrn) - len(tmp)))
        start_index = index + len(p) + (len(pttrn) - len(tmp))

    return ptrns, start_indices


txt = input()
ptrn = input()
wild_card = input()

ptrns, start_indices = create_patterns(ptrn, wild_card)
result = sorted(Aho(txt, ptrns))

c = [0] * len(txt)
for i in result:
    index = i[0] - 1 - start_indices[i[1] - 1]
    if (index >= 0) and (index < len(c)):
```

```
        c[index] += 1

for i in range(len(c) - len(ptrn) + 1):
    if c[i] == len(ptrns):
        print(i + 1)
```