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: Ахо-Корасик

Студент гр. 0382 _____ Кондратов Ю.А.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург
2022

Цель работы.

Написать две программы. Одну - решающую задачу точного поиска набора образцов, вторую – решающую задачу точного поиска для одного образца с джокером.

Задание.

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

Вход:

Первая строка содержит текст ($1 \leq |T| \leq 100000$).

Вторая - число n ($1 \leq n \leq 3000$), каждая следующая из n строк содержит шаблон из набора.

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 pp

Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p.

Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

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?ab??c? с джокером ?? встречается дважды в тексте xabvccbababcaxxabvccbababcax.

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, $1 \leq |T| \leq 100000$)

Шаблон (P, $1 \leq |P| \leq 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Индивидуальное задание (вариант 5):

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

Выполнение работы.

Для решения обеих задач был реализован алгоритм Ахо-Корасик.

Сначала строится бор по набору паттернов – дерево с корнем в вершине root, каждая вершина подписана некоторой буквой. В каждой вершине есть словарь, в котором ключи – символы соседних вершин, а значения – ссылки на объекты узлов, а также список номеров паттернов во входном списке их содержащем, которые оканчиваются в данной вершине. Каждый узел также содержит суффиксную ссылку.

Суффиксная ссылка для каждой вершины p — это вершина, в которой оканчивается наидлиннейший собственный суффикс строки, соответствующей вершине p. Суффиксные ссылки строятся при инициализации конечного автомата.

Для поиска суффиксных ссылок мы просматриваем все вершины обходом в ширину. Для каждой вершины переходим сначала по суффиксной ссылке предка, и далее по суффкисным ссылкам пока не окажемся в корне, или не найдём вершину, из которой есть нужный переход

Алгоритм Ахо-Корасик использует построенный автомат для поиска всех вхождений в строку паттернов из набора. При посимвольном просмотре строки мы либо увеличиваем длину совпадающей подстроки, переходя к соседней вершине, либо уменьшаем, переходя по суффиксной ссылке. Добавляем в ответ все паттерны, оканчивающиеся в узлах.

Узлы бора реализованы при помощи класса *Node*. Суффиксная ссылка, узлы-потомки и номера оканчивающихся в этом узле паттернов хранятся в полях *suf_link, sub_nodes* и *patterns* соответственно.

Класс *Trie* отвечает за хранение бора, его методы *__create_tree* и *__add_links* отвечают за построение бора и автомата соответственно.

Функция *aho_corasik* реализует одноимённый алгоритм и возвращает выходные данные в соответствии с требованиями задания 1.

Для выполнения задания 2 была дополнительно реализована функция *generate_patterns,* которая разделяет паттерн на подстроки, не содержащие джокеров, и запоминает индексы начала этих подстрок в паттерне в списке *start_indices*.

После того, как найдены все вхождения полученных паттернов, определяется для какого i – индекса в строке, для всех паттернов p_i – p_s = i, где p_i - индекс вхождения паттерна, p_s – стартовый индекс паттерна – эти индексы i и будут индексами вхождениями искомого паттерна.

Для выполнения индивидуального задания в класс *Trie* было добавлено поле (список) *nodes*, хранящее все узлы бора. При помощи прохода по всем узлам находится максимальное количество исходящих из узла дуг.

Для того чтобы вырезать все вхождения паттернов в строку, сначала все символы, находящиеся на позициях паттернов, заменяются на символ, не

входящий в алфавит (например «_»), после чего строка разделяется на подстроки по этому разделителю, пустые подстроки удаляются.

Оценка сложности алгоритмов.

Сложность алгоритма Ахо-Корасик составляет O(a+h+k), где a – суммарная длина подстрок h – длина текста, k – общая длина всех совпадений. Во втором задании ещё добавляется функция generate_patters, однако на асимптотику это не влияет.

Тестирование.

Тестирование производилось при помощи библиотеки pytest.

Рассмотренные при тестировании решения первой задачи случаи представлены в таблице 1.

Таблица 1 – Результаты тестирования решения первой задачи

Входные данные	Выходные данные	Описание	Вердикт
NTAG 3 TAGT TAG T	2 2 2 3	Тест из условия	passed
a 1 b		Граничный случай	passed
abcda 3 abc bc bcda	1 1 2 2 2 3	Паттерны пересекаются	passed

abcabcabcabcabc 1 abc	1 1 4 1 7 1 10 1 13 1	Паттерны не пересекаются	passed
abcdeffedcbadefabc 5 abcdef abc bcd cde def	1 2 2 3 3 4 1 1 4 5 13 5 16 2	Большое количество суффиксных ссылок	passed

Рассмотренные при тестировании решения второй задачи случаи представлены в таблице 2.

Таблица 2 – Результаты тестирования решения второй задачи

Входные данные	Выходные данные	Описание	Вердикт
ACTANCA A$$A$ $	1	Тест из условия	passed
abcdef b *	2	Граничный случай	passed
abcdeffbcbfbdbfbbh ___f___ _	3 4 8 12	Слева и справа есть джокеры	passed

fbabfabbfcd f___f___ _	1	Не хватает длины строки для второго вхождения паттерна	passed
fffabcdeabcdeabcdeff __abcd_abc___bcd__ _	2	Большое количество суффиксных ссылок	passed

Протокол тестирования представлен на рисунке 3.

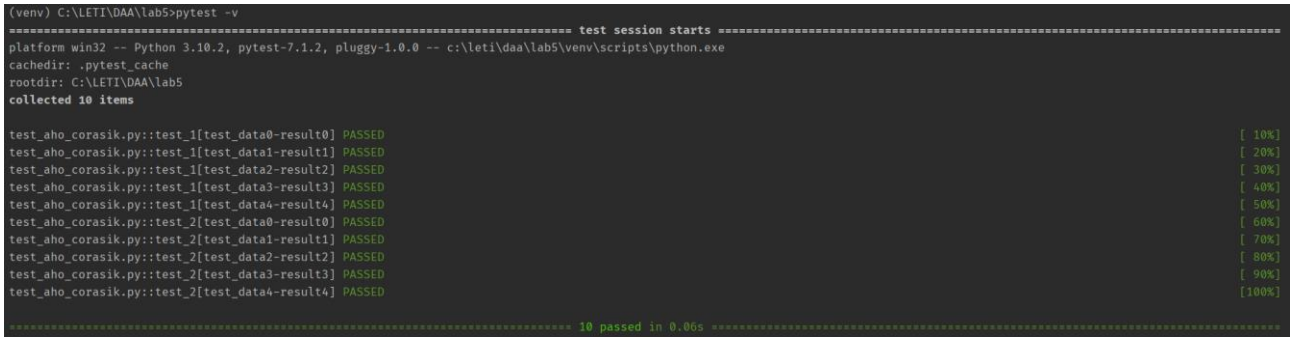

Рисунок 3 – Протокол тестирования

Выводы.

В результате выполнения работы был изучен алгоритм поиска вхождений каждой подстроки из набора в строку – алгоритм Ахо-Корасик. Было изучено понятие бора и конечного автомата. Решены две задачи. Написаны две программы, одна - решающая задачу точного поиска набора образцов, вторая – решающая задачу точного поиска для одного образца с джокером.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: tesk1.py

```
class Node:
    def __init__(self, suf_link=None):
        self.sub_nodes = {}
        self.suf_link = suf_link
        self.patterns = []


class Trie:
    def __init__(self, patterns: list[str]):
        self.__create_tree(patterns)
        self.__add_links()

    def __create_tree(self, patterns):
        self.root = Node()
        for ind, pattern in enumerate(patterns):
            node = self.root
            for symbol in pattern:
                node = node.sub_nodes.setdefault(symbol,
Node(self.root))
            node.patterns.append(ind)

    def __add_links(self):
        queue = [x for x in self.root.sub_nodes.values()]

        while queue:
            cur = queue.pop(0)

            for symbol, node in cur.sub_nodes.items():
                queue.append(node)

                link = cur.suf_link
                while not (link is None or symbol in link.sub_nodes):
                    link = link.suf_link

                node.suf_link = link.sub_nodes[symbol] if link else
self.root
                node.patterns += node.suf_link.patterns


def aho_corasick(string, patterns):
    trie = Trie(patterns)
    ans = []
    node = trie.root
    for i in range(len(string)):
        while node is not None and string[i] not in node.sub_nodes:
            node = node.suf_link
        if node is None:
            node = trie.root
            continue
        node = node.sub_nodes[string[i]]
        for pattern in node.patterns:
            ans.append((i - len(patterns[pattern]) + 2, pattern + 1))
```

```
        return ans


    if __name__ == "__main__":
        text = input()
        n = int(input())
        patterns = []
        for i in range(n):
            patterns.append(input())

        result = sorted(aho_corasick(text, patterns))
        for i in result:
            print(i[0], i[1])
```

Название файла: task2.py

```
    # put your python code here


    class Node:
        def __init__(self, suf_link=None):
            self.sub_nodes = {}
            self.suf_link = suf_link
            self.patterns = []


    class Trie:
        def __init__(self, patterns: list[str]):
            self.__create_tree(patterns)
            self.__add_links()

        def __create_tree(self, patterns):
            self.root = Node()
            for ind, pattern in enumerate(patterns):
                node = self.root
                for symbol in pattern:
                    node = node.sub_nodes.setdefault(symbol,
Node(self.root))
                node.patterns.append(ind)

        def __add_links(self):
            queue = [x for x in self.root.sub_nodes.values()]

            while queue:
                cur = queue.pop(0)

                for symbol, node in cur.sub_nodes.items():
                    queue.append(node)

                    link = cur.suf_link
                    while not (link is None or symbol in link.sub_nodes):
                        link = link.suf_link

                    node.suf_link = link.sub_nodes[symbol] if link else
self.root
                    node.patterns += node.suf_link.patterns
```

```
def aho_corasick(string, patterns):
    trie = Trie(patterns)
    ans = []
    node = trie.root
    for i in range(len(string)):
        while node is not None and string[i] not in node.sub_nodes:
            node = node.suf_link
        if node is None:
            node = trie.root
            continue
        node = node.sub_nodes[string[i]]
        for pattern in node.patterns:
            ans.append((i - len(patterns[pattern]) + 1, pattern))
    return ans


def generate_patterns(pattern, wild_card):
    parts = list(filter(bool, pattern.split(wild_card)))
    start_indices = []
    flag = 1
    for i, c in enumerate(pattern):
        if c == wild_card:
            flag = 1
            continue
        if flag:
            start_indices.append(i)
            flag = 0
    return parts, start_indices


def solve(text, pattern, wild_card):
    patterns, starts = generate_patterns(pattern, wild_card)
    indices = aho_corasick(text, patterns)
    c = [0] * len(text)
    for i, p_i in indices:
        index = i - starts[p_i]
        if 0 <= index < len(c):
            c[index] += 1

    res = []
    for i in range(len(c) - len(pattern) + 1):
        if c[i] == len(patterns):
            res.append(i + 1)
    return res


if __name__ == "__main__":
    txt = input()
    p = input()
    wc = input()
    ans = solve(txt, p, wc)
    print(*ans, sep="\n")
```