

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра МОЭВМ**

**ОТЧЕТ**  
**по лабораторной работе № 5**  
**по дисциплине «Построение и Анализ Алгоритмов»**  
**Тема: «Ахо-Корасик»**

Студент гр. 3343

Преподаватель



Коршков А.А.

Жангиров Т. Р.

Санкт-Петербург

2025

## **Цель работы**

Написать программы на основе алгоритма Ахо-Корасик для нахождения вхождения всех образцов в строке, а также найти индексы вхождения образцов с джокерами.

## **Задания**

### **№1**

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

Вторая - число  $n$  ( $1 \leq n \leq 3000$ ), каждая следующая из  $n$  строк содержит шаблон из набора  $P = \{p_1, \dots, p_n\}$   $1 \leq |p_i| \leq 75$ ,

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:**

NTAG

3

TAGT

TAG

T

#### **Sample Output:**

2 2

2 3

### **№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  $xabvccbababcsax$ .

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 ( $T, 1 \leq |T| \leq 1000000$ )

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.

**Sample Input:**

ACTANCA

A\$\$\$A\$

\$

**Sample Output:**

1

**Задание варианта:**

**№7.** Вывод графического представления автомата.

**Примечания для варианта:**

1) В автомате должны быть и использоваться не только суффиксные ссылки, но и конечные ссылки.

2) Для обоих заданий на программирование должны быть версии кода с выводом промежуточных данных. В них, в частности, должны выводиться построение бора и автомата, построенный автомат (в виде, например, описания каждой вершины автомата), процесс его использования.

## **Основные теоретические положения**

### **Описание алгоритмов:**

Алгоритм создает префиксное дерево из букв искомых подстрок. Вершины, в которых искомая подстрока заканчивается называют терминальной и выделяется специальным цветом в графическом представлении. Суффиксная ссылка вершины  $u$  – это вершина  $v$ , такая что строка  $v$  является максимальным суффиксом строки  $u$ . Для корня и вершин, исходящих из корня, суффиксной ссылкой является корень. Для остальных вершин осуществляется переход по суффиксной ссылке родителя  $u$ , если оттуда есть ребро с заданным символом, суффиксная ссылка назначается в вершину, куда это ребро ведет. Суффиксные ссылки находятся не автоматически для каждой вершины, а вычисляются во время работы программы при обращении к специальному методу.

Текст, в котором нужно найти подстроки побуквенно передается в автомат. Начиная из корня, автомат переходит по ребру, соответствующему переданному символу. Если нужного ребра нет, переходит по ссылке. Если встреченная вершина является терминальной, значит была встречена подстрока. Если найдено совпадение нужно пройти по терминальным ссылкам, если они не None, чтобы вывести все шаблоны заканчивающиеся на этом месте. Номер подстроки хранится в поле `pattern_numbers` вершины. В ответ сохраняются индекс, на котором началась эта подстрока в тексте и сам номер подстроки.

Алгоритм Ахо-Корасик иногда называют «расширенной версией КМП». Схожесть этих алгоритмов заключается в нахождении всех вхождений заданных шаблонов в тексте за линейное время относительно его длины. Однако если в КМП используется префикс-функция для вычисления максимального совпадения префикса и суффикса шаблона (что позволяет избежать полного перебора при несовпадении символов), то в алгоритме Ахо-Корасик применяется бор (дерево строк) и автомат с дополнительными ссылками, что делает его применимым для множественного поиска сразу нескольких шаблонов.

### **Оценка сложности по памяти и операциям**

$O(M \cdot \alpha + N + Z)$  – сложность алгоритма Ахо-Корасик по времени, где

$M$  – суммарный размер всех шаблонов,  $\alpha$  – размер алфавита,  $N$  – длина текста,  $Z$  – количество найденных вхождений. Эта сложность получается из сложности создания автомата с  $O(M \cdot \alpha)$  и сложности поиска в тексте  $O(N + Z)$ .

$O(M \cdot \alpha)$  – сложность алгоритма Ахо-Корасик по памяти, где  $M$  – суммарная длина всех шаблонов,  $\alpha$  – размер алфавита.

Для алгоритма Ахо-Корасик с джокером сложность по времени  $O(M \cdot \alpha + N + Z \cdot K)$ , где  $K$  – количество сегментов в паттерне (без джокеров),  $M$  – суммарная длина всех сегментов паттерна (без джокеров). Сложность по памяти не изменяется.

## Выполнение работы

### Описание работы

Для решения заданий были написаны два класса Vertex и Trie, которые представляют вершину автомата и сам бор.

В классе Vertex описаны следующие методы:

`__init__(self, id_: int, alpha: int, parent: "Vertex" or None = None, pchar: str or None = None) -> None` – конструктор класса вершины Vertex. В качестве аргументов принимает номер вершины, размер алфавита, родительскую вершину (если есть), символ родительской вершины (если есть).

`is_terminal(self) -> bool` – возвращает True, если вершина является терминальной

`@is_terminal.setter`

`is_terminal(self, value: bool) -> None` – позволяет установить флаг для терминальной вершины

`id(self) -> int` – возвращает присвоенный идентификатор вершины

`sufflink(self) -> "Vertex" or None` – возвращает суффиксную ссылку на вершину, если суффиксная ссылка была вычислена для данной вершины.

`@sufflink.setter`

`sufflink(self, value) -> None` - позволяет установить значение для суффиксной ссылки

`parent(self) -> "Vertex" or None` – возвращает родительскую вершину, если это не корень бора.

`pchar(self) -> str or None` – возвращает символ родительской вершины, если это не корень бора.

`__str__(self) -> str` – возвращает информацию в строковом виде для вершины

В классе Trie описаны следующие методы:

`__init__(self, alpha: int = 5) -> None` – конструктор для класса автомата Ахо-Корасик. На вход принимает размер алфавита (по умолчанию 5, для заданного алфавита {A: 0, C: 1, G: 2, T: 3, N: 4})

`size(self) -> int` – возвращает кол-во вершин в дереве  
`last(self) -> Vertex` – возвращает последнюю вершину в дереве  
`alpha(self) -> int` – возвращает размер алфавита  
`vertices(self) -> list[Vertex]` – возвращает список вершин  
`root(self) ->` возвращает корень дерева  
`add(self, s: str, pattern_num: int) -> None` – добавляет образец в дерево  
`search(self, s: str) -> list[tuple[int, int]]` - проверяет, есть ли строка в дереве и возвращает  
`get_link(self, v: Vertex) -> Vertex` – находит и возвращает суффиксную ссылку для вершины.  
`go(self, v: Vertex, char: str) -> Vertex` - Возвращает вершину, в которую ведет переход по символу `char` из вершины `v`.  
`precompute_sufflinks(self) -> None` – предварительно вычисляет все суффиксные ссылки (нужно для визуализации автомата)  
`visualize(self, file_name: str = "aho_corasick") -> None` - Создает графическое представление автомата Ахо-Корасик и сохраняет его в png файл. Также создаётся легенда для графа. На вход принимает имя файла, в который нужно сохранить визуализацию.  
`print_bor_structure(self) -> None` - печатает структуру бора, а именно для каждой вершины пишет наименование, родителя, если вершина терминальная – указывает, каким шаблонам соответствует и возможные пути вниз.  
`print_automaton_structure(self) -> None` - печатает структуру автомата, для каждой вершины отмечает суффиксную ссылку и возможные пути вниз.  
 Также для класса `Trie` есть вспомогательные внутренние методы:  
`_num(c: str) -> int` – возвращает номер буквы в заданном алфавите из 5 букв: {A: 0, C: 1, G: 2, T: 3, N: 4}  
`_char(idx: int) -> int` – возвращает символ буквы, соответствующий номеру в алфавите.  
 В файле `main.py` есть несколько функций для решения заданий:



main() -> None – главная функция, которая запускает функции для решения заданий

aho\_corasick\_search() -> None – функция, которая ищет позиции вхождения всех заданных образцов в тексте.

search\_with\_wildcard() -> None – функция, которая ищет индексы вхождения образца с джокером.

visualize\_and\_print(trie: Trie, filename: str) -> None - выводит информацию о вершинах и графическое представление автомата.

Для понимания графического представления есть наглядная легенда графа.

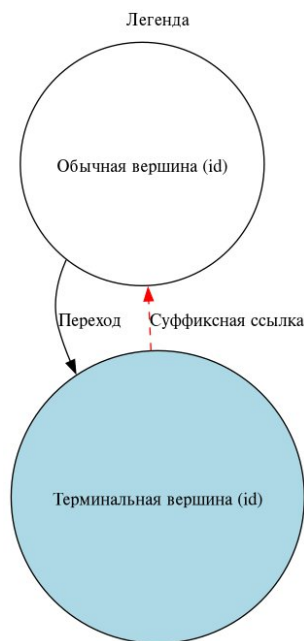
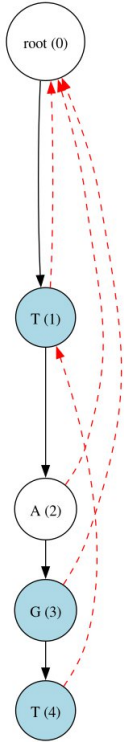


Рисунок 1 – Легенда графа

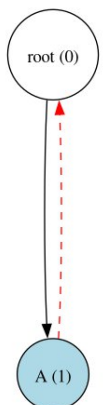
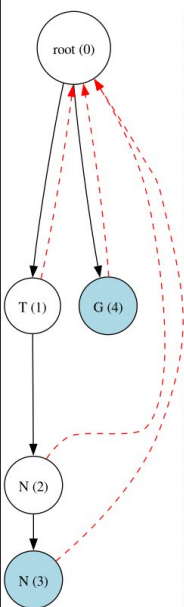
## Тестирование

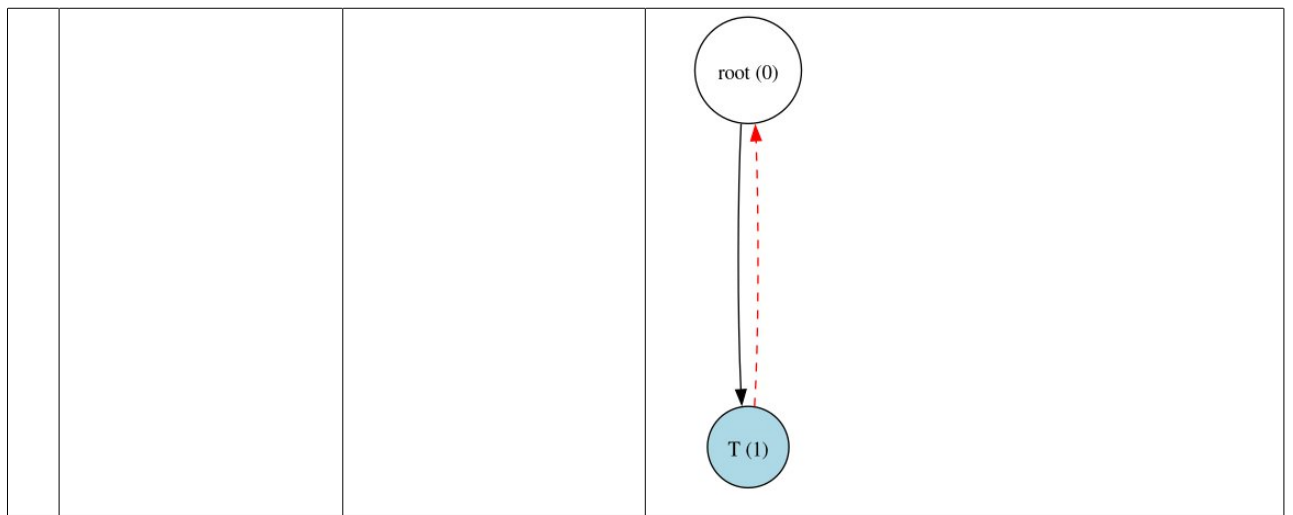
Таблица 1 – Тестирование алгоритмов

№	Входные данные	Выходные данные	Комментарии
1	NTAG 3 TAGT	2 2 2 3	Алгоритм Ахо-Корасик. Терминальные вершины отмечены корректно, суффиксные ссылки также корректны.

	TAG T		Идентификаторы вершин отмечаются корректно. 
ACGT 3 2 ACGT CG GT	1 1 2 2 3 3		Алгоритм Ахо-Корасик. Два последних образца содержатся в одном длинном образце.

3	ACGTNGGTCCG 5 ACG CGT TNG CC A	1 1 1 5 2 2 4 3 9 4	<p>Алгоритм Ахо-Корасик. Алгоритм хорошо справляется с построением автомата для большого количества образцов.</p>

4	<p>ACTANCA</p> <p>AS\$A\$</p> <p>\$</p>	1	<p>Ахо-Корасик с джокером. Символ в маске только А.</p> 
5	<p>ACGTNNTGCA</p> <p>TNNXG</p> <p>X</p>	4	<p>Ахо-Корасик с джокером. Корректно распознаёт маску в виде другой буквы. Дерево содержит в себе части TNN и G из маски.</p> 
6	<p>ACGTNNTGCA</p> <p>T&amp;</p> <p>&amp;</p>	<p>4</p> <p>7</p>	<p>Ахо-Корасик с джокером. Находит несколько вхождений в строке.</p>



## **Выводы**

Изучен принцип работы алгоритма Ахо-Корасик. Написаны программы, корректно решающие задачу поиска набора подстрок в строке, в также программа поиска подстроки с джокером. Также была написана визуализация для автомата Ахо-Корасик в графическом представлении.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
# -*- coding: utf-8 -*-  
"""
```

Главный файл программы.

Вариант 7.

Вывод графического представления автомата.

```
"""
```

```
from modules.trie import Trie
```

```
from modules.vertex import Vertex
```

```
def visualize_and_print(trie: Trie, filename: str) -> None:  
    """
```

Выводит информацию о вершинах и графическое представление автомата.

```
:param trie: Построенный автомат
```

```
:param filename: Имя файла с графическим представлением
```

```
:return: None
```

```
    """
```

```
    trie.print_bor_structure()
```

```
    print("\nВычисление оставшихся суффиксных ссылок")
```

лок

```
    trie.precompute_sufflinks() # вычисление всех суффиксных ссы-
```

```
    trie.print_automaton_structure()
```

ния автомата

```
    trie.visualize(filename) # создание графического представле-
```

```
    # Подсчет и вывод числа вершин
```

```
    print("Количество вершин в автомате:", trie.size)
```

```
def aho_corasick_search() -> None:
```

```
    """
```

Алгоритм Ахо-Корасик для поиска всех образцов в тексте.

```
:return: None
```

```
    """
```

```
    text: str = input().strip() # текст для поиска
```

```
    n: int = int(input()) # количество образцов
```

```
    patterns: list[str] = [] # список образцов
```

```
    lengths: list[int] = [] # длины образцов
```

```
    trie: Trie = Trie() # создание префиксного дерева (Бора)
```

```
    for i in range(n):
```

```
        pattern: str = input().strip() # считывание образца
```

```
        patterns.append(pattern) # добавление образца в список
```

```
        lengths.append(len(pattern)) # добавление длины образца
```

в список

```
    i: int
```

```
    pattern: str
```

```
    for i, pattern in enumerate(patterns):
```

```
        trie.add(pattern, i + 1) # Нумерация шаблонов с 1
```

```
    # Поиск образцов в тексте
```

```
    print("\nНачало поиска в тексте:")
```

найденных образцов

```
    occ: list[tuple[int, int]] = [] # список для хранения
```

```

current: Vertex = trie.root # текущая вершина
for i, char in enumerate(text):
    current: Vertex = trie.go(current, char) # переход по
ребру

    print(f"\nШаг {i + 1}: Символ '{char}'")
    print(f"Текущая вершина: {current.id}")
    print(current)
    v: Vertex = current # текущая вершина
    while v != trie.root: # пока не достигли корня
        if v.is_terminal: # если вершина терминальная
            print(f"\tНайдена терминальная вершина {v.id} с
шаблонами {v.pattern_numbers}")
            for p_num in v.pattern_numbers: # для каждого
номера образца
                start: int = i - lengths[p_num - 1] + 1 #
начало образца
                if start >= 0:
                    occ.append((start + 1, p_num)) # Перево-
дим в 1-based индекс
            v: Vertex = trie.get_link(v) # переход по суффиксной
ссылке

# Сортировка и вывод
occ.sort(key=lambda x: (x[0], x[1]))

visualize_and_print(trie, "aho_corasick_automaton")

print("\nРезультаты поиска:")
pos: int
p: int
for pos, p in occ:
    print(f"Позиция {pos}, образец {p}")

def search_with_wildcard() -> None:
    """
    Поиск с учетом джокера.
    :return: None
    """
    text: str = input().strip() # текст для поиска
    pattern: str = input().strip() # образец с джокером
    wildcard: str = input().strip() # джокер
    len_text, len_pattern = len(text), len(pattern) # длины тек-
ста и образца

    segments: list[tuple[str, int]] = [] # список сегментов
    curr: list[str] = [] # текущий сегмент
    start: int = 0 # начало сегмента
    i: int
    ch: str
    for i, ch in enumerate(pattern):
        if ch == wildcard: # если символ - джокер
            if curr:
                segments.append(("".join(curr), start)) # добав-
ление сегмента в список
                curr: list[str] = [] # очистка текущего сегмента
                start: int = i + 1 # обновление начала сегмента
            else:

```



```

        if not curr:
            start: int = i # обновление начала сегмента
            curr.append(ch) # добавление символа в текущий сег-
мент
        if curr:
            segments.append("".join(curr), start) # добавление по-
следнего сегмента в список

        trie: Trie = Trie() # создание префиксного дерева (Бора)
        print("\nСегменты для поиска:")
        pid: int
        seg: str
        off: int
        for pid, (seg, off) in enumerate(segments):
            print(f"Сегмент {pid + 1}: '{seg}' начинается с позиции
{off + 1}")
            trie.add(seg, pid) # добавление сегмента в префиксное
дерево

        occ: list[tuple[int, int]] = trie.search(text) # поиск об-
разцов в тексте
        needed: int = len(segments) # количество сегментов
        counts: list[int] = [0] * (len_text - len_pattern + 1 if
len_text >= len_pattern else 0) # инициализация счетчиков

        end_pos: int
        pid: int
        for end_pos, pid in occ:
            seg: str
            off: int
            seg, off = segments[pid] # получение сегмента и его сме-
щения
            print(f"Сегмент '{seg}' (PID {pid}) найден на позиции
{end_pos - len(seg) + 1}-{end_pos}")
            l: int = len(seg) # длина сегмента
            p: int = end_pos - (off + 1 - 1) # вычисление позиции
            if 0 <= p <= len_text - len_pattern:
                counts[p] += 1 # увеличение счетчика для позиции p
        visualize_and_print(trie, "aho_corasick_wildcard_automaton")

        print("\nПозиции с полным совпадением:")
        i: int
        c: int
        for i, c in enumerate(counts):
            if c == needed:
                print(i + 1) # вывод позиций, где все сегменты
найденны

def main() -> None:
    """
    Главная функция программы.
    :return: None
    """
    print("Задание #1: Нахождение всех образцов в тексте")
    aho_corasick_search()

```

```

        print("Задание #2: Решение задачи точного поиска одного об-
разца с джокером")
        search_with_wildcard()

if __name__ == '__main__':
    main()
Название файла: vertex.py
# -*- coding: utf-8 -*-
"""
Класс Vertex, представляющий вершину в автомате.
"""

class Vertex:
    """
    Класс, представляющий вершину в автомате.
    """

    def __init__(self, id_: int, alpha: int,
                  parent: "Vertex" or None = None, pchar: str or
None = None) -> None:
        """
        Конструктор класса Vertex.
        :param id_: Идентификатор вершины.
        :param alpha: Количество символов в алфавите.
        :param parent: Родительская вершина.
        :param pchar: Символ, по которому произошел переход в эту
вершину.
        """
        self.__id: int = id_ # Идентификатор вершины
        self.next: list[Vertex | None] = [None] * alpha # Список
переходов
        self.__is_terminal: bool = False # Флаг, указывающий,
заканчивается ли в этой вершине строка
        self.__parent: Vertex or None = parent # Родительская
вершина
        self.__pchar: str or None = pchar # Символ, по которому
произошел переход в эту вершину
        self.__sufflink: Vertex or None = None # Суффиксная
ссылка
        self.go: list[Vertex | None] = [None] * alpha # Список
переходов по символам
        self.pattern_numbers: list = [] # Номера шаблонов, за-
канчивающихся здесь

    @property
    def is_terminal(self) -> bool:
        """
        Возвращает True, если заканчивается в этой вершине обра-
зец.
        :return: True, если заканчивается, иначе False.
        """
        return self.__is_terminal

    @is_terminal.setter
    def is_terminal(self, value: bool) -> None:

```

```

        """
        Устанавливает флаг окончания образца.
        :param value: True, если заканчивается в этой вершине об-
разец, иначе False.
        :return: None
        """
        self.__is_terminal = value

    @property
    def id(self) -> int:
        """
        Возвращает идентификатор вершины.
        :return: Идентификатор вершины.
        """
        return self.__id

    @property
    def sufflink(self) -> "Vertex" or None:
        """
        Возвращает суффиксную ссылку.
        :return: Суффиксная ссылка.
        """
        return self.__sufflink

    @sufflink.setter
    def sufflink(self, value) -> None:
        """
        Устанавливает суффиксную ссылку.
        :param value: Суффиксная ссылка.
        :return: None
        """
        self.__sufflink = value

    @property
    def parent(self) -> "Vertex" or None:
        """
        Возвращает родительскую вершину.
        :return: Родительская вершина.
        """
        return self.__parent

    @property
    def pchar(self) -> str or None:
        """
        Возвращает символ, по которому произошел переход в эту
вершину
        (символ родительской вершины).
        :return: Символ, по которому произошел переход в эту вер-
шину.
        """
        return self.__pchar

    def __str__(self) -> str:
        parent_id = self.parent.id if self.parent else "None"
        sufflink_id = self.sufflink.id if self.sufflink else
"None"
        next_ids = [(i, v.id) for i, v in enumerate(self.next) if
v is not None]

```

```

        go_ids = [(i, v.id) for i, v in enumerate(self.go) if v
is not None]
        return (f"Vertex(id={self.id}, pchar='{self.pchar}',
parent={parent_id}, "
                f"is_terminal={self.is_terminal},
pattern_numbers={self.pattern_numbers}, "
                f"sufflink={sufflink_id}, next={next_ids},
go={go_ids})")

```

**Название файла: trie.py**

```

# -*- coding: utf-8 -*-

```

```

"""

```

```

Класс Trie для работы с префиксными деревьями.

```

```

"""

```

```

from graphviz import Digraph

```

```

from modules.vertex import Vertex

```

```

def _num(c: str) -> int:

```

```

    """

```

```

    Функция для получения номера буквы в алфавите.

```

```

    :param c: Буква

```

```

    :return: Номер буквы

```

```

    """

```

```

    alphabet: dict[str, int] = {

```

```

        'A': 0,

```

```

        'C': 1,

```

```

        'G': 2,

```

```

        'T': 3,

```

```

        'N': 4

```

```

    }

```

```

    return alphabet[c]

```

```

def _char(idx: int) -> str:

```

```

    """

```

```

    Функция для получения буквы по номеру.

```

```

    :param idx: Номер буквы

```

```

    :return: Буква

```

```

    """

```

```

    return ['A', 'C', 'G', 'T', 'N'][idx]

```

```

class Trie:

```

```

    """

```

```

    Класс Trie для работы с префиксными деревьями.

```

```

    """

```

```

    def __init__(self, alpha: int = 5) -> None:

```

```

        """

```

```

        Конструктор класса Trie.

```

```

        :param alpha: Размер алфавита бора.

```

```

        """

```

```

        self.__alpha: int = alpha

```

```

        self.__vertices: list[Vertex] = [Vertex(0, alpha)]

```

```

        self.__root: Vertex = self.vertices[0]

```

```

@property
def size(self) -> int:
    """
    Возвращает количество вершин в дереве.
    :return: Количество вершин в дереве.
    """
    return len(self.vertices)

@property
def last(self) -> Vertex:
    """
    Возвращает последнюю вершину в дереве.
    :return: Последняя вершина в дереве.
    """
    return self.vertices[-1]

@property
def alpha(self) -> int:
    """
    Возвращает размер алфавита.
    :return: Размер алфавита.
    """
    return self.__alpha

@property
def vertices(self) -> list[Vertex]:
    """
    Возвращает список вершин в дереве.
    :return: Список вершин в дереве.
    """
    return self.__vertices

@property
def root(self) -> Vertex:
    """
    Возвращает корень дерева.
    :return: Корень дерева.
    """
    return self.__root

def add(self, s: str, pattern_num: int) -> None:
    """
    Добавляет образец в дерево.
    :param s: Образец строки для добавления.
    :param pattern_num: Номер шаблона.
    :return: None
    """
    v: Vertex = self.root
    print(f"\nДобавление образца '{s}' (номер {pattern_num})")
    char: str
    for char in s:
        idx: int = _num(char)
        if v.next[idx] is None:
            print(f"\tСоздана вершина {self.size} для символа '{char}' (родитель {v.id})")
            self.vertices.append(Vertex(self.size, self.alpha,
v, char))
            v.next[idx]: Vertex = self.last

```

```

        v: Vertex = v.next[idx]
        v.is_terminal = True
        v.pattern_numbers.append(pattern_num)
        print(f"\tВершина {v.id} помечена как терминальная для ша-
блонов {pattern_num}")

    def search(self, s: str) -> list[tuple[int, int]]:
        """
        Проверяет, есть ли строка в дереве.
        :param s: Строка для поиска.
        :return: Список кортежей (позиция, номер шаблона), где по-
зиция - это индекс в строке s
        """
        res: list[tuple[int, int]] = []
        v: Vertex = self.root
        print(f"\nНачало поиска в строке: '{s}'")

        i: int
        char: str
        for i, char in enumerate(s):
            print(f"\nШаг {i + 1}: Символ '{char}' (позиция {i +
1}))")

            v: Vertex = self.go(v, char)
            print(f"Текущая вершина: {v.id}")
            u: Vertex = v
            while u is not self.root:
                if u.is_terminal:
                    print(f"\tНайдена терминальная вершина {u.id}
(шаблоны: {u.pattern_numbers})")
                    pid: int
                    for pid in u.pattern_numbers:
                        res.append((i, pid))
                    print(f"\tПереход по суффиксной ссылке из {u.id}
-> {self.get_link(u).id}")
                    u: Vertex = self.get_link(u)
            print("\nПоиск завершен. Найдено совпадений:", len(res))
            return res

    def get_link(self, v: Vertex) -> Vertex:
        """
        Возвращает суффиксную ссылку для вершины v.
        :param v: Вершина, для которой нужно получить суффиксную
ссылку.
        :return: Суффиксная ссылка для вершины v.
        """
        if v.sufflink is None:
            if self.root in (v, v.parent):
                print(f"\tСуффиксная ссылка вершины {v.id} установ-
лена на корень")
                v.sufflink = self.root
            else:
                print(f"\tВычисление суффиксной ссылки для {v.id}:
через родителя {v.parent.id} и символ '{v.pchar}'")
                v.sufflink = self.go(self.get_link(v.parent), v.pchar)
                print(f"\tВершина {v.id}: суффиксная ссылка ->
{v.sufflink.id}")
            return v.sufflink

```

```

def go(self, v: Vertex, char: str) -> Vertex:
    """
    Возвращает вершину, в которую ведет переход по символу char
    из вершины v.
    :param v: Вершина, из которой нужно сделать переход.
    :param char: Символ, по которому нужно сделать переход.
    :return: Вершина, в которую ведет переход по символу char
    из вершины v.
    """
    idx: int = _num(char)
    v.go[idx]: Vertex
    if v.go[idx] is None:
        if v.next[idx] is not None:
            print(f"\tПрямой переход из {v.id} по '{char}' ->
{v.next[idx].id}")
            v.go[idx]: Vertex = v.next[idx]
        elif v == self.root:
            print(f"\tКорневой переход из {v.id} по '{char}'
-> корень")
            v.go[idx]: Vertex = self.root
        else:
            print(f"\tРекурсивный переход из {v.id} по '{char}'
через суффиксную ссылку")
            v.go[idx]: Vertex = self.go(self.get_link(v), char)
    return v.go[idx]

def precompute_sufflinks(self) -> None:
    """
    Предварительно вычисляет суффиксные ссылки для всех вершин.
    :return: None
    """
    v: Vertex
    for v in self.vertices[1:]:
        self.get_link(v)

def visualize(self, file_name: str = "aho_corasick") -> None:
    """
    Создает графическое представление автомата Ахо-Корасик и
    сохраняет его в файл.
    :param file_name: Имя файла для сохранения графа (без расши-
    рения).
    :return: None
    """
    dot: Digraph = Digraph(comment="Aho-Corasick Automaton")
    dot.attr(rankdir="TB", fontsize="14") # Вертикальное рас-
    положение графа

    with dot.subgraph(name="cluster_automaton") as automaton:
        # Настройка графа
        automaton.attr(label="Автомат Ахо-Корасик",
style="dotted") # Название, стиль обводки

        # Добавление вершин
        v: Vertex
        for v in self.vertices:
            if v == self.root:
                label: str = "root" + f" ({v.id})"
            else:

```

```

        label: str = (v.pchar if v.pchar is not None
else '') + f" ({v.id})"
        if v.is_terminal:
            automaton.node(str(v.id), label, shape="circle",
                           style="filled", fillcolor="lightblue")
        else:
            automaton.node(str(v.id), label, shape="circle")

# Добавление переходов
v: Vertex
for v in self.vertices:
    next_v: Vertex
    for next_v in v.next:
        if next_v is not None:
            automaton.edge(str(v.id), str(next_v.id))

# Добавление суффиксных ссылок
v: Vertex
for v in self.vertices:
    if v.sufflink is not None and v.sufflink != v:
        automaton.edge(str(v.id), str(v.sufflink.id),
style="dashed", color="red", constraint="false")

with dot.subgraph(name="cluster_legend") as legend:
    # Добавление легенды
    legend.attr(label="Легенда", style="dotted")
    # Пример обычной вершины
    legend.node("legend_node", label="Обычная вершина (id)",
shape="circle")
    # Пример терминальной вершины
    legend.node("legend_terminal", label="Терминальная вер-
шина (id)", shape="circle",
style="filled", fillcolor="lightblue")
    # Пример перехода
    legend.edge("legend_node", "legend_terminal",
label="Переход")
    # Пример суффиксной ссылки
    legend.edge("legend_terminal", "legend_node",
style="dashed",
color="red", label="Суффиксная ссылка")

# Сохранение графа
dot.render(file_name, format="png", cleanup=True, view=True)
print(f"Граф сохранен в файл {file_name}.png")

def print_bor_structure(self) -> None:
    """
    Печатает структуру бора.
    :return: None
    """
    print("\nСтруктура бора:")
    for v in self.vertices:
        parent_id: int = v.parent.id if v.parent else -1
        pchar: str = v.pchar if v.pchar else ''
        transitions: list[str] = []
        idx: int
        next_v: Vertex
        for idx, next_v in enumerate(v.next):

```



```

        if next_v is not None:
            char: str = _char(idx)
            transitions.append(f"'{char}': {next_v.id}")
        trans_str: str = ', '.join(transitions) if transitions
    else 'нет'

        term_info: str = (f", терминальная "
                          f"(шаблоны: {v.pattern_numbers})")
    if v.is_terminal else ""
    print(f"Вершина {v.id}: родитель {parent_id}, "
          f"символ '{pchar}'{term_info}, переходы:
{trans_str}")

def print_automaton_structure(self) -> None:
    """
    Печатает структуру автомата (суффиксные ссылки и переходы).
    :return: None
    """
    print("\nСтруктура автомата (суффиксные ссылки и перехо-
ды):")

    for v in self.vertices:
        suff_id: int = v.sufflink.id if v.sufflink else -1
        go_trans: list[str] = []
        idx: int
        go_v: Vertex
        for idx, go_v in enumerate(v.go):
            if go_v is not None:
                char: str = _char(idx)
                go_trans.append(f"'{char}': {go_v.id}")
        go_str: str = ', '.join(go_trans) if go_trans else 'нет'
        print(f"Вершина {v.id}: суффиксная ссылка -> {suff_id},
переходы: {go_str}")

```

Название файла: requirements.txt

pylint

graphviz