**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 |  | Коршков А.А. |
| Преподаватель |  | 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 ≤ ∣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:**

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 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 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≤∣T∣≤100000)

Шаблон (P,1≤∣P∣≤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 называет 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 и, если оттуда есть ребро с заданным символом, суффиксная ссылка назначается в вершину, куда это ребро ведет. Cуффиксные 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 сохранятся 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 · α + N + Z) – сложность алгоритма Ахо-Корасик по времени, где

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

O(M · α) – сложность алгоритма Ахо-Корасик по памяти, где M – суммарная длина всех шаблонов, α - размер алфавита.

Для алгоритма Ахо-Корасик с джокером сложность по времени O(M · α + N + Z· 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 – констурктор 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

tlink(self) -> "Vertex" or None – возвращает терминальную ссылку, если она вычислена и существует для этой вершины.

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 – ввозвращает 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, а также добавляет терминальную ссылку, если она есть у 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 и терминальную 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 файла 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.

Изображение выглядит как текст, круг, снимок экрана, диаграмма

Контент, сгенерированный ИИ, может содержать ошибки.

Рисунок 1 – Легенда графа

# Тестирование

Таблица 1 – Тестирование алгоритмов

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарии |
| 1 | NTAG  3  TAGT  TAG  T | 2 2  2 3 | Алгоритм Ахо-Корасик. Терминальные вершины и суффиксные ссылки отмечены корректно, суффиксные ссылки также корректны. Идентификаторы вершин отмечаются корректно. |
| 2 | ACGT  3  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 | Алгоритм Ахо-Корасик. Алгоритм хорошо справляется с построением автомата для большого количества образцов. В данном примере нет терминальных ссылок. |
| 4 | ACTANCA  A$$A$  $ | 1 | Ахо-Корасик с джокером. Символ в маске только A. |
| 5 | ACGTNNTGCA  TNNXG  X | 4 | Ахо-Корасик с джокером. Корректно распознаёт маску в виде другой буквы. Дерево содержит в себе части TNN и G из маски. |
| 6 | ACGTNNTGCA  T&  & | 4  7 | Ахо-Корасик с джокером. Находит несколько вхождений в строке. |

# 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")

trie.precompute\_sufflinks() # вычисление всех суффиксных ссылок

trie.print\_automaton\_structure()

trie.visualize(filename) # создание графического представления автомата

# 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:")

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

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)) # Переводим в 1-based индекс

# Используем терминальную ссылку вместо полного перебора суффиксных ссылок

print(f"\tПереход по терминальной ссылке: {v.id}." if v != trie.root else "Переход в корень.")

v: Vertex = v.tlink if v.tlink is not None else trie.root

# 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 текста 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)) # добавление 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 сегмент

if curr:

segments.append(("".join(curr), start)) # добавление последнего 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 образцов 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 смещения

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 + l - 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 найдены

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 образца 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.\_\_tlink: Vertex or None = None # Терминальная ссылка

self.go: list[Vertex | None] = [None] \* alpha # Список переходов по символам

self.pattern\_numbers: list = [] # Номера шаблонов, заканчивающихся здесь

@property

def is\_terminal(self) -> bool:

"""

Возвращает True, если заканчивается в этой вершине образец.

: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 образец, 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: "Vertex" or None) -> None:

"""

Устанавливает суффиксную ссылку.

:param value: Суффиксная ссылка.

:return: None

"""

self.\_\_sufflink = value

@property

def tlink(self) -> "Vertex" or None:

"""

Возвращает терминальную ссылку.

:return: Терминальная ссылка.

"""

return self.\_\_tlink

@tlink.setter

def tlink(self, value: "Vertex" or None) -> None:

"""

Устанавливает терминальную ссылку.

:param value: Терминальная ссылка.

:return: None

"""

self.\_\_tlink = 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 вершину.

"""

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"

tlink\_id = self.tlink.id if self.tlink 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}, tlink={tlink\_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

print(f"\tВершина {v.id} помечена как терминальная для шаблонов {pattern\_num}")

v.pattern\_numbers.append(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 позиция - 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))

u: Vertex = u.tlink if u.tlink is not None else self.root

print(f"\tПереход по терминальной ссылке: {u.id}." if u != self.root else "Переход в корень.")

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} установлена 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}")

# Установка терминальной ссылки:

v.tlink = v.sufflink if v.sufflink.is\_terminal else v.sufflink.tlink

# Вывод информации о терминальной ссылке

if v.tlink:

print(f"\tТерминальная ссылка вершины {v.id} установлена на {v.tlink.id}")

else:

print(f"\tТерминальная ссылка вершины {v.id} отсутствует")

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)

node: Vertex = v.go[idx]

if node.sufflink is None:

# Вызов “ленивого” get\_link сам установит sufflink и tlink

self.get\_link(node)

return node

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 расширения).

:return: None

"""

dot: Digraph = Digraph(comment="Aho-Corasick Automaton")

dot.attr(rankdir="TB", fontsize="18", fontname="Arial") # Вертикальное расположение 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 терминальных ссылок

v: Vertex

for v in self.vertices:

if v.tlink is not None:

automaton.edge(str(v.id), str(v.tlink.id), style="dotted",

color="blue", constraint="false")

# Добавление суффиксных ссылок

v: Vertex

for v in self.vertices:

if v.sufflink is not None and v.sufflink != v and v.sufflink != v.tlink:

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", fontname="Arial", margin="20")

# Пример обычной вершины

legend.node("legend\_node", label="Обычная\nвершина (id)", shape="circle",

width="0.5", height="0.3", fontsize="11")

# Пример терминальной вершины

legend.node("legend\_terminal", label="Терминальная\nвершина (id)", shape="circle",

style="filled", fillcolor="lightblue", width="0.5",

height="0.3", fontsize="11")

# Пример перехода

legend.edge("legend\_node", "legend\_terminal", label="Переход", fontsize="11")

# Пример суффиксной ссылки

legend.edge("legend\_terminal", "legend\_node", style="dashed",

color="red", label="Суффиксная\nссылка", fontsize="11")

# Пример терминальной ссылки

legend.edge("legend\_terminal", "legend\_node", style="dotted",

color="blue", label="Терминальная\nссылка", fontsize="11")

# 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 (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 переходы):")

for v in self.vertices:

suff\_id: int = v.sufflink.id if v.sufflink else -1

tlink\_id = v.tlink.id if v.tlink 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}, "

f"терминальная ссылка -> {tlink\_id}, переходы: {go\_str}")

Название файла: requirements.txt

pylint

graphviz