**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. 3343 |  | Наумкин А.Д. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

**Цель работы.**

Изучить принцип работы алгоритма Кнута-Морриса\_Пратта. Написать функцию, вычисляющую для каждого элемента строки максимальное значение длины префикса и с помощью данной функции решить поставленные задачи. А именно написать программу, осуществляющую поиск вхождений подстроки в строку, а также программу, определяющую, являются ли строки циклическим сдвигом друг друга, найти индекс начала вхождения второй строки в первую.

**Задание №1.**

Разработайте программу,  решающую задачу точного поиска набора образцов.  
  
**Вход:**  
Первая строка содержит текст (T,1≤∣T∣≤100000*T*,1≤∣*T*∣≤100000 ).  
Вторая - число n*n* (1≤n≤30001≤*n*≤3000), каждая следующая из n*n* строк содержит шаблон из набора P={p1,…,pn}1≤∣pi∣≤75*P*={*p*1​,…,*pn*​}1≤∣*pi*​∣≤75  
Все строки содержат символы из алфавита {A,C,G,T,N}{*A*,*C*,*G*,*T*,*N*}  
**Выход:**  
Все вхождения образцов из P*P* в T*T*.  
Каждое вхождение образца в текст представить в виде двух чисел - i *i*  p*p*  
Где i*i* - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p*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

**Задание №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*P* необходимо найти все вхождения РР в текст ТТ.  
  
Например, образец аb??с?а*b*??с? с джокером ?? встречается дважды в тексте *xabvccbababcax*.  
  
Символ джокер не входит в алфавит, символы которого используются в T*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}{*A*,*C*,*G*,*T*,*N*}  
  
**Вход:**  
Текст (T,1≤∣T∣≤100000*T*,1≤∣*T*∣≤100000 )  
Шаблон (P,1≤∣P∣≤40*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

**Вариант 1**. На месте джокера может быть любой символ, за исключением заданного.

**Описание алгоритмов.**

**Описание алгоритма Ахо-Корасик.**

Алгоритм создает префиксное дерево из букв искомых подстрок. Затем в полученном дереве ищутся суффиксные ссылки. 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. Далее создаются терминальные ссылки – такие суффиксные ссылки, которые ведут в вершину, которая является терминальной.

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 (подстрок) хранится в поле *terminate* вершины. В ответ сохранятся индекс, на котором началась эта подстрока в тексте и сам номер подстроки.

**Сложность по времени:**

Т.к при построении префиксного дерева запускается цикл по длине

каждой подстроки (суммарная длина подстрок - n), и из каждой вершины

может исходить максимум k ребер (где k – размер алфавита), то построение

префиксного дерева происходит за O(n\*k)

Алгоритм в цикле проходит по тексту длины s: O(s)

Итого: O(n\*k + s)

**Сложность по памяти:**

Алгоритм создает префиксное дерево с n вершинами, каждая вершина

хранит массив вершин, инцидентных ей, размером k (k – размер алфавита).

Итого: O(n\*k) Описание алгоритма для нахождения шаблонов с маской.

**Описание модифицированного алгоритма.**

Алгоритм тот же, но в качестве подстрок берутся кусочки шаблона, разделенные джокером, запоминаются позиции полученных подстрок в исходном шаблоне. Создается массив С длины s, где s – длина текста, где ищется шаблон. При нахождении подстроки, в массиве С увеличивается на единицу число по индексу, соответствующему возможному началу шаблона. Индекс высчитывается по формуле: текущий индекс - (длина найденной подстроки - 1) - (позиция подстроки в шаблоне -1). Затем проходим по полученному массиву, каждый i для которого С[i] = количеству подстрок, является вероятным началом шаблона. В соответствии с индивидуализацией, для каждого найденного шаблона проверяются буквы, стоящие на месте джокера. Если не было встречено запрещенного символа, найденный шаблон добавляется в ответ.

**Сложность по времени для модифицированного алгоритма:**

Затраты по времени такие же как в обычном алгоритме, но дополнительно проход по массиву С длины s: Итого: O(n\*k + s + s) = O(n\*k + s + t)

**Сложность по памяти для модифицированного алгоритма:**

Затраты по памяти такие же как в обычном алгоритме, но дополнительно создается массив С длины s. Затраты по памяти О(n\*k + s)

**Описание функций.**

1. **Класс Node**: Представляет узел в префиксном дереве (trie), содержащий ссылки на родителя, детей, суффиксные и терминальные ссылки, а также информацию о терминальности узла и его имени.
2. **Класс Trie**: Реализует префиксное дерево для хранения шаблонов, с методами для построения дерева, создания суффиксных и терминальных ссылок, и поиска шаблонов в тексте с использованием алгоритма Ахо-Корасик.
3. **Метод \_create\_trie(self) -> None**: Создает префиксное дерево на основе переданных шаблонов, добавляя узлы для каждого символа шаблона и отмечая терминальные узлы.
4. **Метод `\_create\_suffix\_link\_for\_node(self, node: Node) -> None**: Создает суффиксную ссылку для конкретного узла, используя суффиксные ссылки его родителя и других узлов.
5. **Метод \_create\_suffix\_links(self) -> None**: Обходит дерево в ширину и создает суффиксные ссылки для всех узлов, начиная с корня.
6. **Метод \_create\_terminal\_links(self) -> None**: Создает терминальные ссылки для узлов, которые ведут к ближайшему терминальному узлу по суффиксным ссылкам.
7. **Метод Aho\_Korasik(self, text: str) -> list[str]:** Реализует алгоритм Ахо-Корасик для поиска всех вхождений шаблонов в тексте, используя суффиксные и терминальные ссылки.
8. **Функция get\_text() -> str:** Запрашивает у пользователя входной текст для поиска шаблонов.
9. **Функция get\_patterns() -> dict:** Запрашивает у пользователя количество шаблонов и сами шаблоны, возвращая их в виде словаря с уникальными идентификаторами.
10. **Функция main() -> None:** Основная функция, которая запрашивает текст и шаблоны, создает дерево и выполняет поиск шаблонов в тексте, выводя результаты.

**Тестирование.**

|  |  |  |
| --- | --- | --- |
| Входные данные | Ответ | Комментарий |
| NTAG  3  TAGT  TAG  T | 2 2  2 3 | Верно |
| ACCGTACA  2  AC  GT | 1 1  4 2  6 1 | Верно |
| ACGT  3  ACGT  CG  GT | 1 1  2 2  3 3 | Верно |

Таблица 1 – Тестирование алгоритма Ахо-Корасик

|  |  |  |
| --- | --- | --- |
| Входные данные | Ответ | Комментарий |
| ACTANCA  A$$A$  $  G | 1 | Верно |
| ACACAA  ACXA  X  Y | 3 | Верно |
| ACGANGAAAT  A$G  $  C | 4 | Верно |

Таблица 1 – Тестирование алгоритма поиска с джокером

Результат работы программы с отладочным выводом для первого задания (см. рис 1, 2, 3).

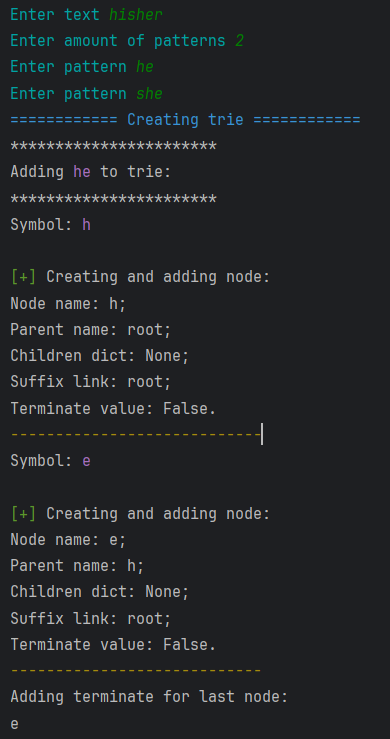


Рисунок 1 – Добавление слова в бор



Рисунок 2 – Создание суффиксных и терминальных ссылок

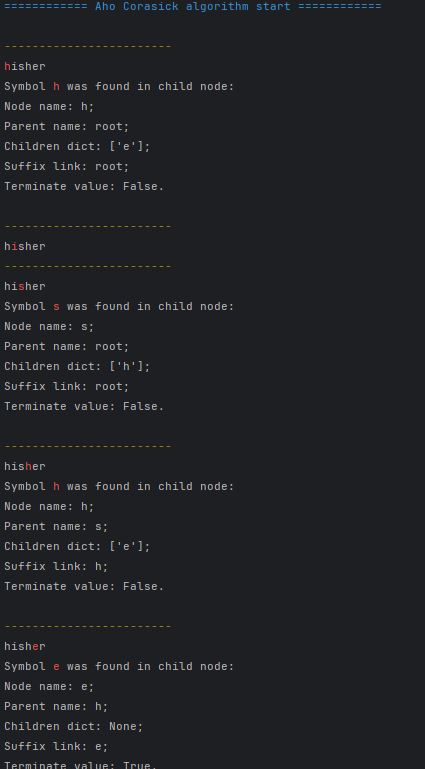


Рисунок 3 – Процесс работы алгоритма Ахо-Корасик

Пример отличающегося вывода для второй программы (остальные логи аналогичны заданию 1)

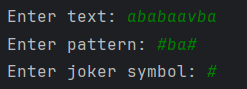


Рисунок 4 – Исходные данные

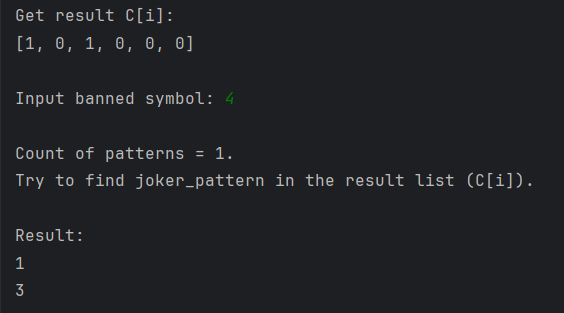


Рисунок 5 – Итоговый вывод

Также была написана программа для визуализации бора

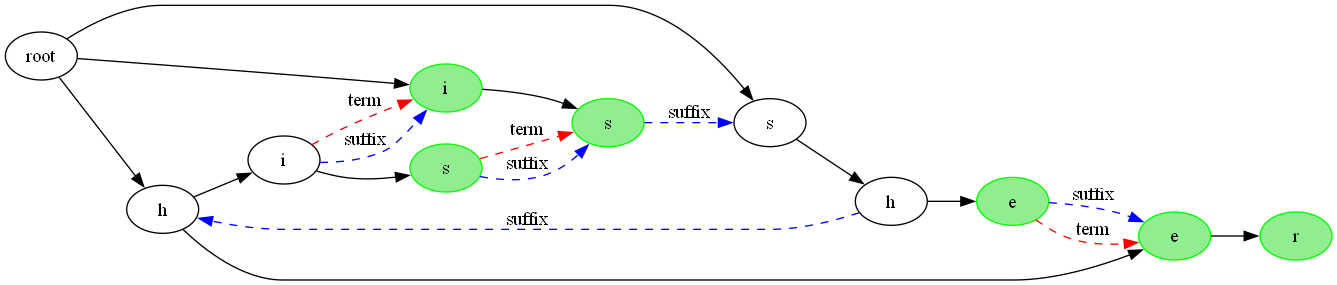


Рисунок 6 – вывод программы для словаря  
{"her": 1, "she": 2, "his": 3,"is":4,"i":5,"he":6}

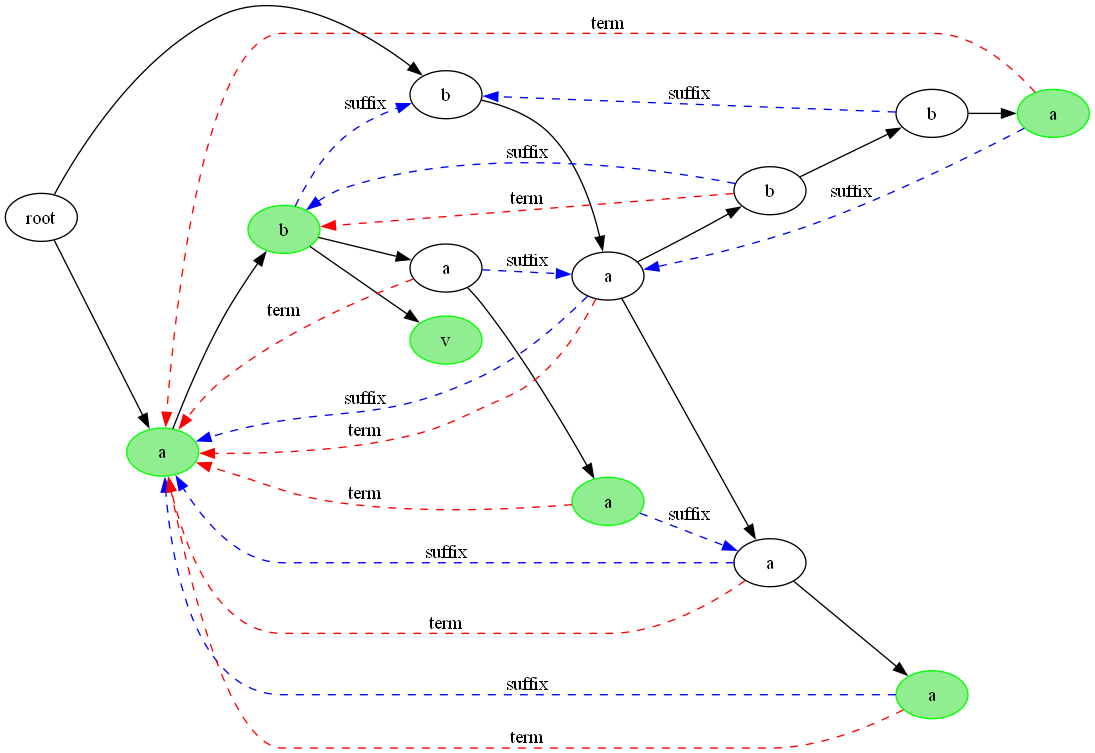


Рисунок 7 – вывод программы для словаря  
{"abaa": 1, "baaa": 2, "abv": 3,"babba":4,"ab":5,"a":6}

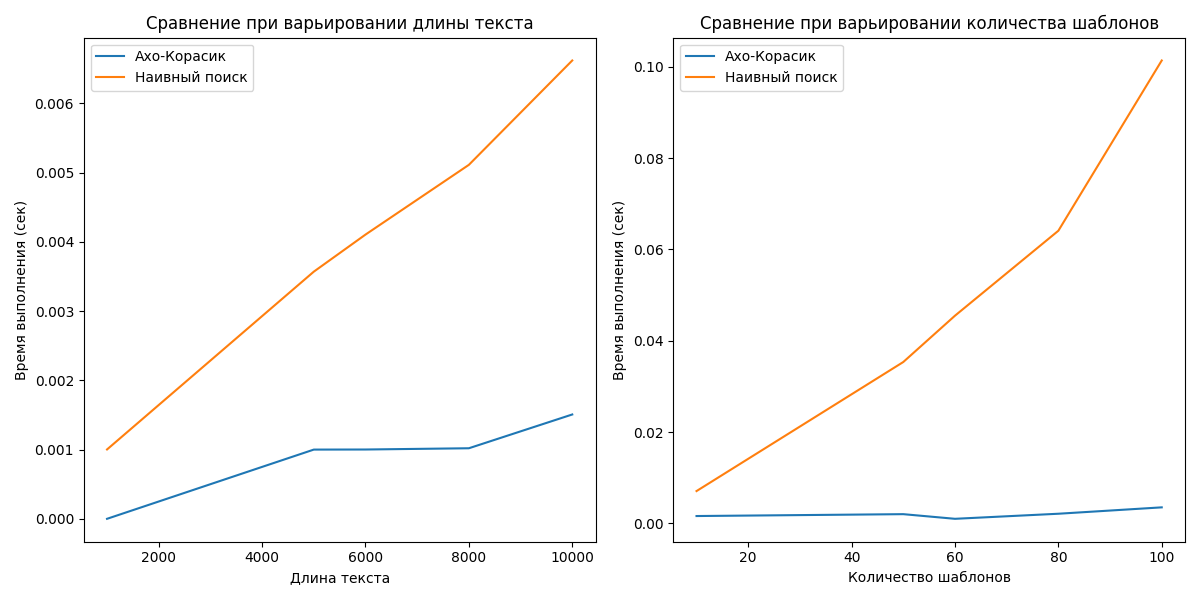
**Исследование.**

Рисунок 8 – Сравнение алгоритма Ахо-Корасик и наивного поиска

Можно сделать вывод, что Ахо-Корасик выполняется значительно быстрее, чем наивный алгоритм.

**Выводы.**

Изучен принцип работы алгоритма Ахо-Корасик. Написаны программы, корректно решающие задачу поиска набора подстрок в строке, в также программа поиска подстроки с джокером.

# **ПРИЛОЖЕНИЕ А**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Название файла:joker.py

from operator import itemgetter

from termcolor import colored

from colors import Colors

# Узел дерева

class Node:

def \_\_init\_\_(self, link=None, name="root"):

self.parent = None

self.children: dict[str, Node] = {}

self.suffix\_link = link

self.terminate = 0 # Номер шаблона, если узел терминальный

self.name = name # Имя узла (символ или "root")

self.deep: int = 0 # Глубина узла

def \_\_str\_\_(self):

return (

f"Node name: {self.name};\n"

f"Parent name: {self.parent.name if self.parent else None};\n"

f"Children dict: {list(self.children.keys()) if self.children.keys() else None};\n"

f"Suffix link: {self.suffix\_link.name if self.suffix\_link else None};\n"

f"Terminate value: {True if self.terminate else False};\n"

f"Deep value: {self.deep}."

)

# Создание дерева

def create\_tree(patterns: dict) -> Node:

root = Node()

list\_patterns = list(patterns.keys())

print(Colors.blue("=== Creating trie ==="))

for i in range(len(list\_patterns)):

node = root

print(f"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\nAdding {Colors.magenta(list\_patterns[i])} to trie:\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for symbol in list\_patterns[i]:

print(f"Symbol: {Colors.magenta(symbol)}\n")

if symbol not in node.children.keys():

temp\_node = Node(name=symbol)

temp\_node.deep = node.deep + 1

node.children[symbol] = temp\_node

temp\_node.parent = node

node = temp\_node

print(f"{Colors.green('[+] ')}Creating and adding node:\n{temp\_node}\n----------------------------")

else:

print(f"Already have this symbol:\n{node.children[symbol]}\n")

node = node.children[symbol]

node.terminate = patterns[list\_patterns[i]]

print(f"Adding terminate for last node:\n{node}\n")

return root

# Создание суффиксных ссылок

def create\_suffix\_links(root: Node) -> None:

print(Colors.blue("\n=== Making Suffix Links ===\n"))

queue = []

for child in root.children.values():

child.suffix\_link = root

queue.append(child)

print(f"Set suffix link for node '{child.name}' -> root")

while queue:

cur\_node = queue.pop(0)

print(f"\nProcessing parent node: '{cur\_node.name}'")

for child in cur\_node.children.values():

queue.append(child)

symbol = child.name

link = cur\_node.suffix\_link

print(f"\n Processing child: '{child.name}'")

print(f" Current suffix link of parent '{cur\_node.name}': '{link.name if link else 'None'}'")

while link and (symbol not in link.children.keys()):

print(

f" Symbol '{symbol}' not found in children of '{link.name}'. Moving to suffix link: '{link.suffix\_link.name if link.suffix\_link else 'None'}'")

link = link.suffix\_link

if link:

child.suffix\_link = link.children[symbol]

print(f" Set suffix link for node '{child.name}' -> '{link.children[symbol].name}'")

else:

child.suffix\_link = root

print(f" Set suffix link for node '{child.name}' -> root")

def \_create\_terminal\_links(root) -> None:

queue = [x for x in root.children.values()]

while queue:

cur\_node = queue.pop(0)

temp = cur\_node

for child in cur\_node.children.values():

queue.append(child)

while temp.name != "root":

if temp.terminate and temp != cur\_node:

cur\_node.terminal\_link = temp

break

temp = temp.suffix\_link

# Функция с алгоритмом Ахо-Корасик

def aho\_corasick() -> list[str]:

text = get\_text()

pattern\_input, joker, patterns, len\_patt = get\_pattern()

if patterns:

tree = create\_tree(patterns)

create\_suffix\_links(tree)

\_create\_terminal\_links(tree)

print(Colors.blue("\n=== Aho Korasik algorithm start === \n"))

result = [0] \* len(text)

node = tree

for index in range(len(text)):

colored\_text = text[:index] + Colors.red(text[index]) + text[index + 1:]

print(colored\_text)

print(f"Current symbol '{text[index]}'\n")

while node and (text[index] not in node.children.keys()):

node = node.suffix\_link

if node:

node = node.children[text[index]]

print(f"Symbol was found in child node:\n{node}\n")

temp = node

while temp:

if temp.terminate:

pattern = text[index - temp.deep + 1: index + 1]

print(

f"Get terminate value for \"{pattern}\" at index = {index - temp.deep + 2}. Pattern number is {temp.terminate}.\n")

for j in patterns[pattern]:

if (index\_j := index - temp.deep - j + 1) >= 0:

result[index\_j] += 1

temp = temp.suffix\_link

else:

node = tree

k = sum([len(elem) for elem in list(patterns.values())])

print(f"Get result C[i]:\n{result[:len(result) - len\_patt + 1]}\n")

ban\_symbol: str = input("Input banned symbol: ")

if len(ban\_symbol) != 1:

raise ValueError("Invalid ban symbol!")

print(f"\nCount of patterns = {k}.\nTry to find joker\_pattern in the result list (C[i]).\n")

output = []

for i in range(len(result) - len\_patt + 1):

if k == result[i]:

find\_ban = []

text\_patt = text[i: i + len\_patt]

for j in range(len\_patt):

if pattern\_input[j] == joker:

find\_ban.append(text\_patt[j])

if ban\_symbol not in find\_ban:

output.append(str(i + 1))

return output

def get\_text() -> str:

return input("Enter text: ")

def get\_pattern() -> (str, str, dict, int):

pattern = input("Enter pattern: ")

joker = input("Enter joker symbol: ")

patterns = get\_sub\_patterns(pattern, joker)

return pattern, joker, patterns, len(pattern)

def get\_sub\_patterns(pattern: str, joker: str) -> dict[str, list[int]]:

patterns: dict[str, list[int]] = {}

j = -1

for i in range(len(pattern)):

if pattern[i] == joker:

if j < i - 1:

s = pattern[j + 1: i]

if s not in patterns.keys():

patterns[s] = []

patterns[s].append(j + 1)

j = i

if j != len(pattern) - 1:

s = pattern[j + 1:]

if s not in patterns.keys():

patterns[s] = []

patterns[s].append(j + 1)

return patterns

# Основная функция

def main():

result = aho\_corasick()

if result:

print("Result:\n" + '\n'.join(result))

else:

print("No such pattern in the text.")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Название файла: aho\_korasik.py

import copy

from operator import itemgetter

from termcolor import colored

from colors import Colors

# Узел дерева

class Node:

def \_\_init\_\_(self, link=None, name="root"):

self.parent = None

self.children: dict[str, Node] = {}

self.suffix\_link = link

self.terminal\_link=None

self.terminate = 0 # Номер шаблона, если узел терминальный

self.name = name # Имя узла (символ или "root")

def \_\_str\_\_(self):

return (

f"Node name: {self.name};\n"

f"Parent name: {self.parent.name if self.parent else None};\n"

f"Children dict: {list(self.children.keys()) if self.children.keys() else None};\n"

f"Suffix link: {self.suffix\_link.name if self.suffix\_link else None};\n"

f"Terminate value: {True if self.terminate else False}."

)

class Trie:

def \_\_init\_\_(self, patterns: dict):

self.root = Node()

self.patterns = patterns

self.terminate\_patterns = dict(zip(patterns.values(), patterns.keys()))

self.\_create\_trie()

self.\_create\_suffix\_links()

self.\_create\_terminal\_links()

# Создание дерева

def \_create\_trie(self):

print(Colors.blue("=== Creating trie ==="))

list\_patterns = list(self.patterns.keys())

for pattern in list\_patterns:

node = self.root

print(f"\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\nAdding {Colors.magenta(pattern)} to trie:\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*")

for symbol in pattern:

print(f"Symbol: {Colors.magenta(symbol)}\n")

if symbol not in node.children:

temp\_node = Node(link=self.root,name=symbol)

node.children[symbol] = temp\_node

temp\_node.parent = node

node = temp\_node

print(f"{Colors.green('[+] ')}Creating and adding node:\n{temp\_node}\n----------------------------")

else:

print(f"Already have this symbol:\n{node.children[symbol]}\n")

node = node.children[symbol]

# Устанавливаем terminate для последнего символа шаблона

node.terminate = self.patterns[pattern]

print(f"Adding terminate for last node:\n{node.name}\n")

# Создание суффиксных ссылок

def \_create\_suffix\_link\_for\_node(self, node):

if node == self.root:

return

link = node.parent.suffix\_link

print(f"\n Processing node: '{node.name}'")

print(f" Current suffix link of parent '{node.parent.name}': '{link.name if link else 'None'}'")

while link and (node.name not in link.children.keys()):

print(

f" Symbol '{node.name}' not found in children of '{link.name}'. Moving to suffix link: '{link.suffix\_link.name if link.suffix\_link else 'root'}'")

link = link.suffix\_link

if link:

node.suffix\_link = link.children[node.name]

print(f" Set suffix link for node '{node.name}' -> '{link.children[node.name].name}'")

def \_create\_suffix\_links(self):

print(Colors.blue("\n=== Making Suffix Links ===\n"))

queue = [x for x in self.root.children.values()]

while queue:

cur\_node = queue.pop(0)

print(f"\nProcessing parent node: '{cur\_node.name}'")

for child in cur\_node.children.values():

queue.append(child)

self.\_create\_suffix\_link\_for\_node(child)

def \_create\_terminal\_links(self):

print(Colors.blue("\n=== Making Terminal Links ===\n"))

queue = [x for x in self.root.children.values()]

while queue:

cur\_node = queue.pop(0)

temp=cur\_node

for child in cur\_node.children.values():

queue.append(child)

while temp.name != "root":

if temp.terminate and temp!=cur\_node:

cur\_node.terminal\_link=temp

print(f"\nTerminal link for node: '{cur\_node.name}' -> {temp.name}")

break

temp=temp.suffix\_link

def Aho\_Korasik(self, text: str) -> list[str]:

print(Colors.blue("\n=== Aho Korasik algorithm start === \n"))

result = []

node = self.root

for index in range(len(text)):

colored\_text = text[:index] + Colors.red(text[index]) + text[index+1:]

print(colored\_text)

while node and (text[index] not in node.children.keys()):

node = node.suffix\_link

if node:

node = node.children[text[index]]

print(f"Symbol was found in child node:\n{node}\n")

temp = node

while temp:

if temp.terminate:

print(

f"Get terminate value for \"{self.terminate\_patterns[temp.terminate]}\" "

f"at index = {index - len(self.terminate\_patterns[temp.terminate]) + 2}. "

f"Pattern number is {temp.terminate}.\n"

)

result.append([index - len(self.terminate\_patterns[temp.terminate]) + 2,temp.terminate])

temp = temp.terminal\_link

else:

node = self.root

result = sorted(result, key=itemgetter(0, 1))

result = [' '.join(map(str, elem)) for elem in result]

return result

def get\_text() -> str:

return input("Enter text ")

def get\_patterns() -> dict:

n = int(input("Enter amount of patterns "))

patterns: dict = {}

for pattern\_n in range(n):

pattern = input("Enter pattern ")

patterns[pattern] = pattern\_n + 1

return patterns

# Основная функция

def main():

text = get\_text()

patterns = get\_patterns()

#patterns = {"her": 1, "she": 2, "his": 3,"is":4,"i":5,"he":6}

trie = Trie(patterns)

result = trie.Aho\_Korasik(text)

print('\n'.join(result))

if \_\_name\_\_ == "\_\_main\_\_":

main()

Название файла: trie\_visualiser.py

from graphviz import Digraph

from aho\_korasik import Trie

class TrieVisualizer:

def \_\_init\_\_(self, trie):

self.trie = trie

self.graph = Digraph(comment="Trie Visualization", format="png")

self.graph.attr(rankdir="LR")

def \_add\_node(self, node, parent\_name=None):

node\_name = f"{node.name}\_{id(node)}"

if node.terminate:

self.graph.node(node\_name, label=node.name, color="green", style="filled", fillcolor="lightgreen")

else:

self.graph.node(node\_name, label=node.name)

if parent\_name:

self.graph.edge(parent\_name, node\_name)

if node.suffix\_link and node.suffix\_link != self.trie.root:

suffix\_name = f"{node.suffix\_link.name}\_{id(node.suffix\_link)}"

self.graph.edge(node\_name, suffix\_name, style="dashed", color="blue", label="suffix")

if node.terminal\_link and node.terminal\_link != self.trie.root:

terminal\_link\_name = f"{node.terminal\_link.name}\_{id(node.terminal\_link)}"

self.graph.edge(node\_name, terminal\_link\_name, style="dashed", color="red", label="term")

for child in node.children.values():

self.\_add\_node(child, node\_name)

def visualize(self, filename="trie"):

self.\_add\_node(self.trie.root)

self.graph.render(filename, cleanup=True)

print(f"Дерево сохранено в файл {filename}.png")

#patterns = {"her": 1, "she": 2, "his": 3,"is":4,"i":5,"he":6}

patterns = {"abaa": 1, "baaa": 2, "abv": 3,"babba":4,"ab":5,"a":6}

trie = Trie(patterns)

visualizer = TrieVisualizer(trie)

visualizer.visualize(filename="trie\_visualization")

Название файла: Colors.py

class Colors:

RED = "\033[31m"

GREEN = "\033[32m"

YELLOW = "\033[33m"

BLUE = "\033[34m"

MAGENTA = "\033[35m"

CYAN = "\033[36m"

WHITE = "\033[37m"

RESET = "\033[0m"

@staticmethod

def red(text):

return f"{Colors.RED}{text}{Colors.RESET}"

@staticmethod

def green(text):

return f"{Colors.GREEN}{text}{Colors.RESET}"

@staticmethod

def yellow(text):

return f"{Colors.YELLOW}{text}{Colors.RESET}"

@staticmethod

def blue(text):

return f"{Colors.BLUE}{text}{Colors.RESET}"

@staticmethod

def magenta(text):

return f"{Colors.MAGENTA}{text}{Colors.RESET}"

@staticmethod

def cyan(text):

return f"{Colors.CYAN}{text}{Colors.RESET}"

@staticmethod

def white(text):

return f"{Colors.WHITE}{text}{Colors.RESET}"