**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: Поиск набора подстрок в строке. Ахо-Корасик.

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Никишин С.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

**Цель работы.**

Изучить алгоритм поиска набора подстрок в строке. Разработать программу, находящую все вхождения подстрок в строку с помощью алгоритма Ахо-Корасика.

**Задание.**

Вариант 2. Подсчитать количество вершин в автомате; вывести список найденных образцов, имеющих пересечения с другими найденными образцами в строке поиска.

**Задание 6.1.**

Разработайте программу,  решающую задачу точного поиска набора образцов.  
  
**Вход:**  
Первая строка содержит текст (*T*, 1 ≤ ∣*T*∣ ≤ 100000 ).  
Вторая - число *n* (1 ≤ *n* ≤ 3000), каждая следующая из *n* строк содержит шаблон из набора *P* = {*p*1​,…,*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

**Задание 6.2.**

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 а*b*??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

**Описание алгоритма.**

Алгоритм Ахо-Корасика — это эффективный алгоритм поиска множества подстрок в тексте. Он сочетает в себе идеи использования борa (префиксного дерева) и автомата.

Этапы алгоритма (используется в заданиях 6.1 и 6.2):

1. Построение бора (Trie). Создаём бор для всех искомых строк:

1.1. Корень соответствует пустой строке.

1.2. Для каждого символа строки переходим по соответствующему ребру (если его нет — создаем).

1.3. В конце помечаем терминальную вершину (храним длину образца или его идентификатор).

2. Построение суффиксных и конечных ссылок:

2.1. Строим суффиксные ссылки (перебираем узлы-детей и их символы):

2.1.1. Суффиксная ссылка корня и суффиксная ссылка его потомков указывает на сам корень.

2.1.2. Иначе идем по суффиксным ссылкам родителя, пока не найдем узел-ребенка с текущим символом.

2.2. Построение конечных ссылок:

2.2.1. Если суффиксная ссылка ведет в терминальный узел, она является терминальной

2.2.2. Иначе продолжаем идти по суффиксным ссылкам, пока не найдем терминальный узел.

3. Поиск шаблонов в тексте:

3.1. Начинаем с корня.

3.2. Для каждого символа в тексте:

3.2.1. Пока символ и символ в рассматриваемом узле не совпали, идем по суффиксным ссылкам.

3.2.2. Если символы совпали, переходим в дочерний узел.

3.3. Проверяем является ли данный узел терминальным и идем п конечным ссылкам.

3.4. Сохраняем найденные вхождения.

4. В итоге получаем найденные вхождения шаблонов.

Поиск при использовании шаблона с символом-джокером (используется в задании 6.2):

1. Делим шаблон на подшаблоны игнорируя символы-джокеры.

2. Ищем вхождения подшаблонов алгоритмом Ахо-Корасика.

3. Ищем вхождения шаблона:

3.1. Создается массив, который хранит количество найденных подстрок для каждой позиции текста.

3.2. Перебираем позиции найденных вхождений подшаблонов. Увеличиваем значение ячейки массива с индексов позиции вхождения.

3.3. Если число в ячейки массива соответствует числу подшаблонов, индекс ячейки – начало вхождения полного шаблона.

Описание алгоритма индивидуализации:

1. Ищем вхождения шаблонов алгоритмом Ахо-Корасика.

2. Рассматриваем найденный массив вхождений. Рассматриваем каждое вхождения, с остальными, с которыми еще не рассматривалось это вхождение ранее.

3. Вычисляем концы рассматриваемых вхождений и проверяем не оканчивается ли хотя бы одно из них раньше, чем начинается другое.

3.1. Если оканчивается рассматриваем следующую пару

3.2. Иначе находим точку пересечения и заносим информацию о вхождениях и точке пересечения в массив найденных пересечений

4. В конце получаем массив найденных пересечений.

|  |  |
| --- | --- |
|  | Сложность по времени |
| Построение бора | O(L) |
| Построение ссылок | O(A \* L) |
| Поиск | O(T + L + n) |
| Деление шаблона на подшаблоны | O(P) |
| Поиск пересечений | O(n2) |

|  |  |
| --- | --- |
|  | Сложность по памяти |
| Хранение автомата | O(A \* L) |

, где L = Σ длин всех шаблонов, T – длина текста, A – число символов в алфавите (число разных символов среди всех шаблонов), P – длина шаблона с джокерами, n – число вхождений

Исходные коды реализующие алгоритмы из заданий 6.1 и 6.2 представлены в приложении (приложении 1 и 2 соответственно).

**Описание функций и структур данных.**

Структуры данных:

1. *TrieNode* – структура, представляющая из себя узел бора.

Содержит поля:

*id* - уникальный идентификатор узла (для вывода)

*isTerminal* - флаг, является ли вершина терминалом

*patternIndices* - номера шаблонов, заканчивающихся в этом узле

*patternLength* - длина шаблона (для определения позиции)

*childrens* - указатели на дочерние узлы

*suffixLink* - суффиксная ссылка

*finalLink* - конечная ссылка

2. *AhoCorasicAlgorithm* – класс, который хранит автомат (бор с ссылками) реализует методы для работы с ним, а так же сам алгоритм Ахо-Корасика.

3. *C = [0] \* (len(text) + 1)* - массив, где C[i] - количество встретившихся в тексте безмасочных подстрок шаблона, который начинается в тексте на позиции i.

Функции и методы:

1. *def \_\_init\_\_(self, patterns: list):* метод класса *AhoCorasicAlgorithm*, инициализирует алгоритм, создает автомат.

Содержит:

*root* – корень бора (тип *TrieNode*)

*nodeCount* – счетчик числа узлов в боре (тип *int*)

Параметры:

*patterns* – массив шаблонов, добавляемых в бор (тип *list*).

Сложность по времени: O(L), где L = Σ длин всех шаблонов

2. *def \_\_add(self, pattern: str, index: int):* метод класса *AhoCorasicAlgorithm*, добавляет новый шаблон в бор.

Параметры:

*pattern* – массив шаблон, добавляемый в бор (тип *str*).

*index* – индекс добавляемого шаблона.

Сложность по времени: O(L), где L = Σ длин всех шаблонов

3. *def \_\_makeLinks(self):* метод класса *AhoCorasicAlgorithm*, cоздает суффиксные и конечные ссылки.

Сложность по времени: O(A \* L), где L = Σ длин всех шаблонов, A – число символов в алфавите (число разных символов среди всех шаблонов)

4. *def search(self, text):* метод класса *AhoCorasicAlgorithm*, ищет все вхождения шаблонов в тексте.

Параметры:

*text* – текст, в котором осуществляется поиск (тип *str*).

Возвращает:

results – отсортированный по позициям в текст массив найденных вхождений (элемент массива имеет вид (позиция начала вхождения, индекс шаблона) – (*int, int*))

Сложность по времени: O(T + L + n), где L = Σ длин всех шаблонов, T – длина текста, n – число вхождений

5. *def getNodeCount(self):* метод класса *AhoCorasicAlgorithm*, позволяет получить число узлов в автомате.

Возвращает:

*nodeCount* – счетчик числа узлов в боре (тип *int*)

Сложность по времени: O(1)

6. *def \_\_printAutomat(self):* метод класса *AhoCorasicAlgorithm*, выводит информацию о построенном автомате.

Сложность по времени: O(L), где L = Σ длин всех шаблонов

7. *def findIntersectingPatterns(results: list, patterns: list):* ищет пересечения вхождений.

Параметры:

*results* –массив найденных вхождений (тип *list*).

*patterns* – массив шаблонов

Возвращает:

intersectionPairs – массив с найденными пересечениями(тип *list*).

Сложность по времени: O(n2), где n – число элементов массива *results.*

8. *def findSubPatterns(pattern: str, jokerSymbol: str):* делит шаблон с символом-джокером на отдельные части.

Параметры:

*pattern* – шалон с джокер-символами, который требуется разделить (тип *str*).

*jokerSymbol* – джокер-символ (тип *str*).

Возвращает:

*subPatterns, positions* – список подшаблонов и список их позиций (типы *list*)

Сложность по времени: O(P), где P – длина шаблона с джокерами.

**Тестирование.**

Тестирование программ представлено на таблице.

Таблица 1. Задание 6.1

|  |  |  |  |
| --- | --- | --- | --- |
| № Теста | Входные данные | Выходные данные | Комментарий |
| 1 | NTAG  3  TAGT  TAG  T | 2 2  2 3 | Верно |
| 2 | shershe  4  he  she  his  hers | 1 2  2 1  2 4  5 2  6 1 | Верно |
| 3 | TTTTTTTTTT  2  NAN  GNG |  | Верно |
| 4 | TATTATAT  3  TATT  TAT  TATTAT | 1 1  1 2  1 3  4 2  6 2 | Верно |
| 5 | ABBBAABAB  3  ABB  ABB  ABA | 1 1  1 2  6 3 | Верно |

Таблица 2. Задание 6.2

|  |  |  |  |
| --- | --- | --- | --- |
| № Теста | Входные данные | Выходные данные | Комментарий |
| 1 | ACTANCA  A$$A$  $ | 1 | Верно |
| 2 | ABBBACBBABBBC  A$BB$  $ | 1  5  9 | Верно |
| 3 | ABBBACBBABBBC  $C$  $ | 5 | Верно |
| 4 | ABBA  C$  $ |  | Верно |

Таблица 3. Индивидуализация

|  |  |  |  |
| --- | --- | --- | --- |
| № Теста | Входные данные | Выходные данные | Комментарий |
| 1 | NTAG  3  TAGT  TAG  T | TAG и T пересекаются в точке 2 | Верно |
| 2 | shershe  4  he  she  his  hers | she и he пересекаются в точке 2  she и her пересекаются в точке 2  he и her пересекаются в точке 2  her и she пересекаются в точке 5  she и he пересекаются в точке 6 | Верно |
| 3 | ABBBACBBABBBC  A$BB$  $ | A$BB$ и A$BB$ пересекаются в точке 5  A$BB$ и A$BB$ пересекаются в точке 9 | Верно |
| 4 | ABCMMMCAB  2  ABC  CAB | Пересечений нет | Верно |

Примеры работ программ с выводом промежуточных результатов при введённом значении указаны в приложении (приложение 3 и приложение 4 для программ 1 и 2 соответственно)

**Выводы.**

Был изучен алгоритм Ахо-Корасика поиска всех вхождений набора подстрок в строку. Разработана программа, реализующая алгоритм Ахо-Корасика, а также алгоритм поиска шаблона с джокерами.

**Приложение**

Приложение 1

from collections import deque

import graphviz

class TrieNode:

"""Класс узла бора"""

def \_\_init\_\_(self, nodeId: int):

self.id = nodeId # уникальный идентификатор узла

self.isTerminal = False # флаг, является ли вершина терминалом

self.patternIndices = [] # номера шаблонов, заканчивающихся в этом узле

self.patternLength = 0 # длина шаблона

self.childrens = {} # указатели на дочерние узлы

self.suffixLink = None # суффиксная ссылка

self.finalLink = None # конечная ссылка

class AhoCorasicAlgorithm:

"""Класс, реализующий алгоритм Ахо-Корасика"""

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

"""Инициализирует алгоритм, создает автомат"""

self.root = TrieNode(0) # корневой узел

self.root.suffixLink = self.root # суффиксная ссылка корня

self.nodeCount = 1 # счетчик узлов

self.patterns = patterns # сохраняем шаблоны

self.max\_arcs = 0 # максимальное количество дуг из одной вершины

# строим бор

for index in range(len(patterns)):

self.\_\_add(patterns[index], index)

# создаем ссылки, строим автомат

self.\_\_makeLinks()

def \_\_add(self, pattern: str, index: int):

"""Добавляет новый шаблон в бор"""

currentNode = self.root

# перебираем символы шаблона

for char in pattern:

if char not in currentNode.childrens:

newNode = TrieNode(self.nodeCount)

currentNode.childrens[char] = newNode

self.nodeCount += 1

# Обновляем максимальное количество дуг

self.max\_arcs = max(self.max\_arcs, len(currentNode.childrens))

currentNode = currentNode.childrens[char]

currentNode.isTerminal = True

currentNode.patternIndices.append(index)

currentNode.patternLength = len(pattern)

def \_\_makeLinks(self):

"""Создает суффиксные и конечные ссылки"""

queue = deque()

queue.append(self.root)

while queue:

currentNode = queue.popleft()

for char, childNode in currentNode.childrens.items():

queue.append(childNode)

# для детей корня суффиксная ссылка ведет в корень

if currentNode == self.root:

childNode.suffixLink = self.root

else:

# ищем первую возможную суффиксную ссылку

temp = currentNode.suffixLink

while (temp != self.root) and (char not in temp.childrens):

temp = temp.suffixLink

# устанавливаем найденную ссылку или ссылку на корень

if char in temp.childrens:

childNode.suffixLink = temp.childrens[char]

else:

childNode.suffixLink = self.root

# построение конечной ссылки

if childNode.suffixLink.isTerminal:

childNode.finalLink = childNode.suffixLink

else:

childNode.finalLink = childNode.suffixLink.finalLink

def search(self, text):

"""Ищет все вхождения шаблонов в тексте"""

results = []

currentNode = self.root

for position in range(len(text)):

char = text[position]

# используем суффиксные ссылки при отсутствии перехода

while (currentNode != self.root) and (char not in currentNode.childrens):

currentNode = currentNode.suffixLink

# переходим по символу, если переход существует

if char in currentNode.childrens:

currentNode = currentNode.childrens[char]

# проверяем терминальные узлы

if currentNode.isTerminal:

for patternIndex in currentNode.patternIndices:

startPosition = position - currentNode.patternLength + 1

results.append((startPosition, patternIndex, currentNode.patternLength))

# проверяем конечные ссылки для нахождения всех вложенных шаблонов

temp = currentNode.finalLink

while temp:

for patternIndex in temp.patternIndices:

startPosition = position - temp.patternLength + 1

results.append((startPosition, patternIndex, temp.patternLength))

temp = temp.finalLink

return sorted(results)

def getNodeCount(self):

"""Позволяет получить число узлов в автомате"""

return self.nodeCount

def getMaxArcs(self):

"""Возвращает максимальное количество дуг из одной вершины"""

return self.max\_arcs

def visualizeBOR(self, filename="bor\_tree"):

"""Визуализирует бор в PNG файл"""

dot = graphviz.Digraph(comment='BOR Tree')

dot.attr('node', shape='circle')

queue = deque([self.root])

visited = set([self.root])

# Добавляем корневой узел

root\_label = "0"

if self.root.isTerminal:

root\_label += "\\n" + ",".join(str(i+1) for i in self.root.patternIndices)

dot.node('0', root\_label, style='filled', fillcolor='lightblue')

while queue:

node = queue.popleft()

for char, child in node.childrens.items():

if child not in visited:

visited.add(child)

queue.append(child)

# Создаем метку для узла

node\_label = f"{child.id}"

if child.isTerminal:

patterns = ",".join(str(i+1) for i in child.patternIndices)

node\_label += f"\\n{patterns}"

dot.node(str(child.id), node\_label)

# Добавляем ребро

dot.edge(str(node.id), str(child.id), label=char)

# Сохраняем в файл

dot.render(filename, format='png', cleanup=True)

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

def printAutomatInfo(self):

"""Выводит краткую информацию об автомате"""

print(f"Информация об автомате:")

print(f"Количество узлов: {self.getNodeCount()}")

print(f"Количество шаблонов: {len(self.patterns)}")

print(f"Максимальное количество дуг из одной вершины: {self.getMaxArcs()}")

# Подсчитываем дополнительную статистику

terminal\_count = 0

queue = deque([self.root])

while queue:

node = queue.popleft()

if node.isTerminal:

terminal\_count += 1

for child in node.childrens.values():

queue.append(child)

print(f"Терминальных узлов: {terminal\_count}")

def removeFoundPatterns(text, results, patterns):

"""Вырезает найденные образцы из строки и возвращает остаток"""

if not results:

return text

# Создаем массив для отметки позиций, которые нужно удалить

remove\_mask = [False] \* len(text)

# Помечаем позиции, которые попадают в найденные образцы

for start\_pos, pattern\_idx, pattern\_length in results:

pattern = patterns[pattern\_idx]

end\_pos = start\_pos + pattern\_length

# Проверяем границы и помечаем позиции для удаления

for i in range(max(0, start\_pos), min(len(text), end\_pos)):

remove\_mask[i] = True

# Собираем остаток строки

remainder = []

for i, char in enumerate(text):

if not remove\_mask[i]:

remainder.append(char)

return ''.join(remainder)

def findPatternRanges(results, patterns):

"""Находит диапазоны найденных образцов для наглядного вывода"""

ranges = []

for start\_pos, pattern\_idx, pattern\_length in results:

pattern = patterns[pattern\_idx]

end\_pos = start\_pos + pattern\_length

ranges.append((start\_pos, end\_pos, pattern))

return sorted(ranges)

def main():

# ввод данных

text = input("Введите текст: ")

num = int(input("Введите количество шаблонов: "))

patterns = [input(f"Введите шаблон {i + 1}: ") for i in range(num)]

# Создаем автомат Ахо-Корасика

ahoCorasicAlgorithm = AhoCorasicAlgorithm(patterns)

# Визуализируем бор (опционально)

try:

ahoCorasicAlgorithm.visualizeBOR("bor\_visualization")

except:

print("Для визуализации бора установите graphviz: pip install graphviz")

# Выводим информацию об автомате

ahoCorasicAlgorithm.printAutomatInfo()

# Пункт 1 варианта 5: Максимальное количество дуг из одной вершины

max\_arcs = ahoCorasicAlgorithm.getMaxArcs()

print(f"\n=== РЕЗУЛЬТАТЫ ДЛЯ ВАРИАНТА 5 ===")

print(f"1. Максимальное количество дуг, исходящих из одной вершины: {max\_arcs}")

# Поиск вхождений

results = ahoCorasicAlgorithm.search(text)

# Вывод результатов поиска

print(f"\nНайдено вхождений: {len(results)}")

if results:

print("Найденные вхождения:")

ranges = findPatternRanges(results, patterns)

for start\_pos, end\_pos, pattern in ranges:

print(f" Позиции {start\_pos}-{end\_pos-1}: '{pattern}'")

# Пункт 2 варианта 5: Вырезаем найденные образцы

remainder = removeFoundPatterns(text, results, patterns)

print(f"\n2. Результат вырезания найденных образцов:")

print(f" Исходный текст: '{text}'")

print(f" Остаток строки: '{remainder}'")

# Дополнительная информация о вырезании

original\_length = len(text)

remainder\_length = len(remainder)

removed\_length = original\_length - remainder\_length

print(f" Статистика: удалено {removed\_length} символов из {original\_length}")

if remainder:

print(f" Сохранено: {remainder\_length} символов")

else:

print(f" Весь текст был удален (состоит только из шаблонов)")

else:

print("Шаблоны не найдены в тексте")

print(f"Остаток строки: '{text}'")

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

main()

Приложение 2

from collections import deque

import graphviz

class TrieNode:

"""Класс узла бора"""

def \_\_init\_\_(self, nodeId: int):

self.id = nodeId # уникальный идентификатор узла

self.isTerminal = False # флаг, является ли вершина терминалом

self.patternIndices = [] # номера шаблонов, заканчивающихся в этом узле

self.patternLength = 0 # длина шаблона

self.childrens = {} # указатели на дочерние узлы

self.suffixLink = None # суффиксная ссылка

self.finalLink = None # конечная ссылка

class AhoCorasicAlgorithm:

"""Класс, реализующий алгоритм Ахо-Корасика"""

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

"""Инициализирует алгоритм, создает автомат"""

self.root = TrieNode(0) # корневой узел

self.root.suffixLink = self.root # суффиксная ссылка корня

self.nodeCount = 1 # счетчик узлов

self.patterns = patterns # сохраняем шаблоны

self.max\_arcs = 0 # максимальное количество дуг из одной вершины

# строим бор

for index in range(len(patterns)):

self.\_\_add(patterns[index], index)

# создаем ссылки, строим автомат

self.\_\_makeLinks()

def \_\_add(self, pattern: str, index: int):

"""Добавляет новый шаблон в бор"""

currentNode = self.root

# перебираем символы шаблона

for char in pattern:

if char not in currentNode.childrens:

newNode = TrieNode(self.nodeCount)

currentNode.childrens[char] = newNode

self.nodeCount += 1

# Обновляем максимальное количество дуг

self.max\_arcs = max(self.max\_arcs, len(currentNode.childrens))

currentNode = currentNode.childrens[char]

currentNode.isTerminal = True

currentNode.patternIndices.append(index)

currentNode.patternLength = len(pattern)

def \_\_makeLinks(self):

"""Создает суффиксные и конечные ссылки"""

queue = deque()

queue.append(self.root)

while queue:

currentNode = queue.popleft()

for char, childNode in currentNode.childrens.items():

queue.append(childNode)

# для детей корня суффиксная ссылка ведет в корень

if currentNode == self.root:

childNode.suffixLink = self.root

else:

# ищем первую возможную суффиксную ссылку

temp = currentNode.suffixLink

while (temp != self.root) and (char not in temp.childrens):

temp = temp.suffixLink

# устанавливаем найденную ссылку или ссылку на корень

if char in temp.childrens:

childNode.suffixLink = temp.childrens[char]

else:

childNode.suffixLink = self.root

# построение конечной ссылки

if childNode.suffixLink.isTerminal:

childNode.finalLink = childNode.suffixLink

else:

childNode.finalLink = childNode.suffixLink.finalLink

def search(self, text):

"""Ищет все вхождения шаблонов в тексте"""

results = []

currentNode = self.root

for position in range(len(text)):

char = text[position]

# используем суффиксные ссылки при отсутствии перехода

while (currentNode != self.root) and (char not in currentNode.childrens):

currentNode = currentNode.suffixLink

# переходим по символу, если переход существует

if char in currentNode.childrens:

currentNode = currentNode.childrens[char]

# проверяем терминальные узлы

if currentNode.isTerminal:

for patternIndex in currentNode.patternIndices:

startPosition = position - currentNode.patternLength + 1

results.append((startPosition, patternIndex, currentNode.patternLength))

# проверяем конечные ссылки для нахождения всех вложенных шаблонов

temp = currentNode.finalLink

while temp:

for patternIndex in temp.patternIndices:

startPosition = position - temp.patternLength + 1

results.append((startPosition, patternIndex, temp.patternLength))

temp = temp.finalLink

return sorted(results)

def getNodeCount(self):

"""Позволяет получить число узлов в автомате"""

return self.nodeCount

def getMaxArcs(self):

"""Возвращает максимальное количество дуг из одной вершины"""

return self.max\_arcs

def visualizeBOR(self, filename="bor\_tree"):

"""Визуализирует бор в PNG файл"""

dot = graphviz.Digraph(comment='BOR Tree')

dot.attr('node', shape='circle')

queue = deque([self.root])

visited = set([self.root])

# Добавляем корневой узел

root\_label = "0"

if self.root.isTerminal:

root\_label += "\\n" + ",".join(str(i+1) for i in self.root.patternIndices)

dot.node('0', root\_label, style='filled', fillcolor='lightblue')

while queue:

node = queue.popleft()

for char, child in node.childrens.items():

if child not in visited:

visited.add(child)

queue.append(child)

# Создаем метку для узла

node\_label = f"{child.id}"

if child.isTerminal:

patterns = ",".join(str(i+1) for i in child.patternIndices)

node\_label += f"\\n{patterns}"

dot.node(str(child.id), node\_label)

# Добавляем ребро

dot.edge(str(node.id), str(child.id), label=char)

# Сохраняем в файл

dot.render(filename, format='png', cleanup=True)

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

def printAutomatInfo(self):

"""Выводит краткую информацию об автомате"""

print(f"Информация об автомате:")

print(f"Количество узлов: {self.getNodeCount()}")

print(f"Количество шаблонов: {len(self.patterns)}")

print(f"Максимальное количество дуг из одной вершины: {self.getMaxArcs()}")

# Подсчитываем дополнительную статистику

terminal\_count = 0

queue = deque([self.root])

while queue:

node = queue.popleft()

if node.isTerminal:

terminal\_count += 1

for child in node.childrens.values():

queue.append(child)

print(f"Терминальных узлов: {terminal\_count}")

def removeFoundPatterns(text, results, patterns):

"""Вырезает найденные образцы из строки и возвращает остаток"""

if not results:

return text

# Создаем массив для отметки позиций, которые нужно удалить

remove\_mask = [False] \* len(text)

# Помечаем позиции, которые попадают в найденные образцы

for start\_pos, pattern\_idx, pattern\_length in results:

pattern = patterns[pattern\_idx]

end\_pos = start\_pos + pattern\_length

# Проверяем границы и помечаем позиции для удаления

for i in range(max(0, start\_pos), min(len(text), end\_pos)):

remove\_mask[i] = True

# Собираем остаток строки

remainder = []

for i, char in enumerate(text):

if not remove\_mask[i]:

remainder.append(char)

return ''.join(remainder)

def findPatternRanges(results, patterns):

"""Находит диапазоны найденных образцов для наглядного вывода"""

ranges = []

for start\_pos, pattern\_idx, pattern\_length in results:

pattern = patterns[pattern\_idx]

end\_pos = start\_pos + pattern\_length

ranges.append((start\_pos, end\_pos, pattern))

return sorted(ranges)

def main():

# ввод данных

text = input("Введите текст: ")

num = int(input("Введите количество шаблонов: "))

patterns = [input(f"Введите шаблон {i + 1}: ") for i in range(num)]

# Создаем автомат Ахо-Корасика

ahoCorasicAlgorithm = AhoCorasicAlgorithm(patterns)

# Визуализируем бор (опционально)

try:

ahoCorasicAlgorithm.visualizeBOR("bor\_visualization")

except:

print("Для визуализации бора установите graphviz: pip install graphviz")

# Выводим информацию об автомате

ahoCorasicAlgorithm.printAutomatInfo()

# Пункт 1 варианта 5: Максимальное количество дуг из одной вершины

max\_arcs = ahoCorasicAlgorithm.getMaxArcs()

print(f"\n=== РЕЗУЛЬТАТЫ ДЛЯ ВАРИАНТА 5 ===")

print(f"1. Максимальное количество дуг, исходящих из одной вершины: {max\_arcs}")

# Поиск вхождений

results = ahoCorasicAlgorithm.search(text)

# Вывод результатов поиска

print(f"\nНайдено вхождений: {len(results)}")

if results:

print("Найденные вхождения:")

ranges = findPatternRanges(results, patterns)

for start\_pos, end\_pos, pattern in ranges:

print(f" Позиции {start\_pos}-{end\_pos-1}: '{pattern}'")

# Пункт 2 варианта 5: Вырезаем найденные образцы

remainder = removeFoundPatterns(text, results, patterns)

print(f"\n2. Результат вырезания найденных образцов:")

print(f" Исходный текст: '{text}'")

print(f" Остаток строки: '{remainder}'")

# Дополнительная информация о вырезании

original\_length = len(text)

remainder\_length = len(remainder)

removed\_length = original\_length - remainder\_length

print(f" Статистика: удалено {removed\_length} символов из {original\_length}")

if remainder:

print(f" Сохранено: {remainder\_length} символов")

else:

print(f" Весь текст был удален (состоит только из шаблонов)")

else:

print("Шаблоны не найдены в тексте")

print(f"Остаток строки: '{text}'")

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

main()