**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. 3388 |  | Ижболдин А.В. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

## Цель работы

Цель данной работы заключается в изучении и реализации алгоритма Ахо-Корасика для многоподстрочного поиска в тексте, а также его модификаций в виде поиска с джокером. Помимо этого подсчет количества символьных и сжатых ссылок.

## Задание

Вариант 3 Вычислить длину самой длинной цепочки из суффиксных ссылок и самой длинной цепочки из конечных ссылок в автомате.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T,1≤∣T∣≤100000T).

Вторая - число nn (1≤n≤3000), каждая следующая из nn 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 pp

(нумерация образцов начинается с 1).

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

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?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∣≤100000T,)

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.

## Выполнение работы

Этот код реализует алгоритм Ахо-Корасика, предназначенный для эффективного поиска множества шаблонов в тексте. Он строит бор (префиксное дерево), в котором каждый узел соответствует состоянию автомата. Каждое состояние имеет переходы по символам, список шаблонов, которые заканчиваются в этом узле, а также ссылки: суффиксную и терминальную, упрощающие обработку при построении автомата и поиске.

После добавления шаблонов в бор происходит построение автомата с помощью обхода в ширину, при этом устанавливаются суффиксные и терминальные ссылки для всех узлов. Основной метод поиска проходит по тексту, переходя из состояния в состояние и фиксируя вхождения шаблонов. Терминальные ссылки позволяют находить и те шаблоны, которые заканчиваются не в текущем узле, а в цепочке суффиксных переходов.

Также реализована возможность поиска с подстановками, где шаблон может содержать спецсимволы (джокеры), обозначающие любое количество или тип символов — для этого шаблон разбивается на подстроки без джокеров и каждая ищется отдельно, после чего проверяется их относительное расположение.

**Класс Node** представляет узел дерева (бора), в котором хранятся переходы к дочерним узлам по символам, список идентификаторов шаблонов, которые заканчиваются в этом узле, суффиксная ссылка на другое состояние автомата и терминальная ссылка — это сокращённая цепочка переходов по суффиксным ссылкам до ближайшего узла с шаблонами. Такие ссылки нужны для оптимизации поиска.

**Класс AhoCorasick** реализует сам алгоритм. Внутри него хранится корневой узел, а также словарь с длинами шаблонов.

Метод **add\_pattern** добавляет новый шаблон в дерево.

Метод **build\_automat** строит суффиксные и терминальные ссылки после добавления всех шаблонов, используя обход в ширину.

Метод **get\_next\_state** определяет следующее состояние автомата при переходе по символу.

Метод **search** осуществляет сам поиск всех шаблонов в переданном тексте, возвращая список всех найденных совпадений с указанием позиции и шаблона. Поиск в алгоритме Ахо–Корасик происходит за один проход по тексту. Алгоритм последовательно читает каждый символ текста и перемещается по построенному автомату (бору с суффиксными и выходными ссылками). Если прямого перехода по текущему символу нет, используется суффиксная ссылка для отката к более короткому префиксу. Когда достигается состояние, соответствующее завершению одного или нескольких шаблонов, они фиксируются как найденные. Благодаря сжатым ссылкам также быстро обрабатываются вложенные шаблоны.

Также внутри класса есть вспомогательные методы для визуализации дерева:

**print\_trie** отображает структуру бора,

**\_print\_node** рекурсивно печатает узлы с отступами,

**\_print\_node\_details** выводит подробности о каждом узле, такие как глубина, ссылки и переходы.

Метод **\_get\_node\_path** восстанавливает путь к узлу от корня.

Ещё два метода — **get\_longest\_suffix\_chain** и **get\_longest\_terminal\_chain** — вычисляют длину самой длинной цепочки суффиксных и терминальных ссылок соответственно.

Функция **search\_with\_wildcard** предназначена для поиска с подстановками. Она разбивает шаблон на подстроки, игнорируя символы-джокеры, строит автомат для этих подстрок, а затем проверяет их вхождения в тексте, учитывая расположение и длину исходного шаблона. Совпадения фиксируются только если все подстроки встали в нужные позиции, соответствующие шаблону с джокерами.

Функции **aho\_default** и **aho\_wildcard** считывают данные из ввода: текст и шаблоны, затем выполняют соответствующий поиск. При необходимости они выводят дополнительную отладочную информацию.

## Асимптотика алгоритма

Построение бора происходит за O(L), где L - сумарная длина шаблонов, так как мы проходим по всем узлам, то построение суффиксных и сжатых ссылок будет занимет еще O(m). По памяти это будет занимать O(L), где L - сумарная длина шаблонов.

Поиск с помощью Ахо-Корасика будет происходить за O(n + k), где k - количество найденых совпадений, n - длина текста, при условии что автомат уже построен.

Для случая с джокером, разбиение на подшаблоны будет занимать O(p), где p - длина шаблона, за O(n + k), мы сможем найти вхождения используя поиск Ахо-Корасика, затем мы должны убедиться что строка подходит под шаблон с джокером, тогда итоговая сложность будет O(n + k\*p).

## Вывод

В ходе лабораторной работы был реализован и проанализирован алгоритм Ахо–Корасик для поиска множества шаблонов в тексте. Алгоритм эффективно справляется с задачей множественного поиска за линейное время по длине текста и суммарной длине шаблонов.

# Исходный код программы

**Название файла: main.py**

from collections import deque

class Node:

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

self.transitions = {} # Переходы по символам к другим узлам

self.pattern\_ids = [] # Список ID шаблонов, заканчивающихся в этом узле

self.suffix\_link = None # Суффиксная ссылка

self.terminal\_link = None # Терминальная ссылка (сжатая суффиксная ссылка)

class AhoCorasick:

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

self.root = Node()

self.patterns\_lengths = {}

def add\_pattern(self, pattern, pattern\_id):

current\_node = self.root

for char in pattern:

if char not in current\_node.transitions:

current\_node.transitions[char] = Node()

current\_node = current\_node.transitions[char]

current\_node.pattern\_ids.append(pattern\_id)

self.patterns\_lengths[pattern\_id] = len(pattern)

def build\_automat(self):

queue = deque()

self.root.suffix\_link = self.root

self.root.terminal\_link = None

for char, child in self.root.transitions.items():

child.suffix\_link = self.root

queue.append(child)

# BFS

while queue:

current = queue.popleft()

for char, child in current.transitions.items():

suffix\_link = current.suffix\_link

while suffix\_link != self.root and char not in suffix\_link.transitions:

suffix\_link = suffix\_link.suffix\_link

if char in suffix\_link.transitions:

child.suffix\_link = suffix\_link.transitions[char]

else:

child.suffix\_link = self.root

if child.suffix\_link.pattern\_ids:

child.terminal\_link = child.suffix\_link

else:

child.terminal\_link = child.suffix\_link.terminal\_link

queue.append(child)

def get\_next\_state(self, state, char):

while state != self.root and char not in state.transitions:

state = state.suffix\_link

if char in state.transitions:

return state.transitions[char]

else:

return self.root

def search(self, text):

result = []

current = self.root

for i, char in enumerate(text):

current = self.get\_next\_state(current, char)

if current.pattern\_ids:

for pattern\_id in current.pattern\_ids:

pattern\_len = self.patterns\_lengths[pattern\_id]

position = i - pattern\_len + 1 + 1

result.append((position, pattern\_id))

node = current.terminal\_link

while node:

for pattern\_id in node.pattern\_ids:

pattern\_len = self.patterns\_lengths[pattern\_id]

position = i - pattern\_len + 1 + 1

result.append((position, pattern\_id))

node = node.terminal\_link

return sorted(result)

def print\_trie(self):

print("=== СТРУКТУРА БОРА ===")

self.\_print\_node(self.root, "", "ROOT")

print("\n=== ДЕТАЛЬНАЯ ИНФОРМАЦИЯ О УЗЛАХ ===")

self.\_print\_node\_details(self.root, "ROOT")

def \_print\_node(self, node, prefix, char\_from\_parent):

pattern\_info = f" [Шаблоны: {node.pattern\_ids}]" if node.pattern\_ids else ""

print(f"{prefix}{char\_from\_parent}{pattern\_info}")

sorted\_chars = sorted(node.transitions.keys())

for i, char in enumerate(sorted\_chars):

child = node.transitions[char]

if i == len(sorted\_chars) - 1:

new\_prefix = prefix + " "

branch = "└── "

else:

new\_prefix = prefix + "│ "

branch = "├── "

self.\_print\_node(child, new\_prefix, f"{branch}{char}")

def \_print\_node\_details(self, node, path, depth=0):

print(f"\nУзел: {path}")

print(f"\tГлубина: {depth}")

print(f"\tШаблоны: {node.pattern\_ids}")

suffix\_path = self.\_get\_node\_path(node.suffix\_link) if node.suffix\_link else "None"

terminal\_path = self.\_get\_node\_path(node.terminal\_link) if node.terminal\_link else "None"

print(f"\tСуффиксная ссылка: {suffix\_path}")

print(f"\tТерминальная ссылка (сжатая суффиксная): {terminal\_path}")

if node.transitions:

print("\tПереходы:")

for char, child in sorted(node.transitions.items()):

child\_path = f"{path} -> {char}"

print(f"\t\t{child\_path}")

for char, child in sorted(node.transitions.items()):

child\_path = f"{path} -> {char}"

self.\_print\_node\_details(child, child\_path, depth + 1)

def \_get\_node\_path(self, node):

if node == self.root:

return "ROOT"

def find\_path(current, target, path=[]):

if current == target:

return path

for char, child in current.transitions.items():

result = find\_path(child, target, path + [char])

if result:

return result

return None

path = find\_path(self.root, node)

if path:

return "ROOT -> " + " -> ".join(path)

else:

return f"Node({id(node)})"

def get\_longest\_suffix\_chain(self):

return self.\_traverse\_and\_check\_suffix\_chains(self.root)

def \_traverse\_and\_check\_suffix\_chains(self, node):

max\_chain\_length = 0

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

current\_chain\_length = self.\_get\_suffix\_chain\_length(child)

max\_chain\_length = max(max\_chain\_length, current\_chain\_length)

child\_max\_length = self.\_traverse\_and\_check\_suffix\_chains(child)

max\_chain\_length = max(max\_chain\_length, child\_max\_length)

return max\_chain\_length

def \_get\_suffix\_chain\_length(self, start\_node):

if start\_node is None:

return 0

visited\_nodes = set()

current = start\_node

chain\_length = 0

while current and current != self.root and current not in visited\_nodes:

visited\_nodes.add(current)

chain\_length += 1

current = current.suffix\_link

return chain\_length

def get\_longest\_terminal\_chain(self):

return self.\_traverse\_and\_check\_terminal\_chains(self.root)

def \_traverse\_and\_check\_terminal\_chains(self, node):

max\_chain\_length = 0

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

current\_chain\_length = self.\_get\_terminal\_chain\_length(child)

max\_chain\_length = max(max\_chain\_length, current\_chain\_length)

child\_max\_length = self.\_traverse\_and\_check\_terminal\_chains(child)

max\_chain\_length = max(max\_chain\_length, child\_max\_length)

return max\_chain\_length

def \_get\_terminal\_chain\_length(self, start\_node):

if start\_node is None or start\_node.terminal\_link is None:

return 0

visited\_nodes = set()

current = start\_node.terminal\_link

chain\_length = 1

while current and current not in visited\_nodes:

visited\_nodes.add(current)

chain\_length += 1

current = current.terminal\_link

if current is None:

break

return chain\_length

def search\_with\_wildcard(text, pattern, wildcard, debug=False):

subpatterns = pattern.split(wildcard)

valid\_subpatterns = []

subpattern\_positions = []

pos = 0

for subpattern in subpatterns:

if subpattern:

valid\_subpatterns.append(subpattern)

subpattern\_positions.append(pos)

pos += len(subpattern) + 1

aho = AhoCorasick()

for i, subpattern in enumerate(valid\_subpatterns, 1):

aho.add\_pattern(subpattern, i)

aho.build\_automat()

if debug:

aho.print\_trie()

# if debug:

# suffix\_chain\_length = aho.get\_longest\_suffix\_chain()

# terminal\_chain\_length = aho.get\_longest\_terminal\_chain()

# print(f"Длина самой длинной цепочки суффиксных ссылок: {suffix\_chain\_length}")

# print(f"Длина самой длинной цепочки терминальных ссылок: {terminal\_chain\_length}")

subpattern\_matches = aho.search(text)

if debug:

print("Позиции подстрок в тексте:", subpattern\_matches)

C = [0] \* (len(text) + 1)

for position, pattern\_id in subpattern\_matches:

start\_pos = position - subpattern\_positions[pattern\_id-1]

if start\_pos > 0:

C[start\_pos] += 1

if debug:

print("Массив C:", C)

pattern\_positions = []

k = len(valid\_subpatterns)

for i in range(1, len(text) - len(pattern) + 2):

if C[i] == k:

match = True

for j in range(len(pattern)):

if i + j - 1 >= len(text):

match = False

break

if pattern[j] != wildcard and pattern[j] != text[i+j-1]:

match = False

break

if match:

pattern\_positions.append(i)

return pattern\_positions

def aho\_default\_test():

text = "abcabeabcabd"

patterns = ["ab", "abc", "aba", "bca", "c", "d"]

print(f"Текст: {text}")

print("Шаблоны:")

for i, pattern in enumerate(patterns, 1):

print(f"{i}. {pattern}")

aho = AhoCorasick()

for i, pattern in enumerate(patterns, 1):

aho.add\_pattern(pattern, i)

print("\nСтруктура бора до построения автомата:")

aho.print\_trie()

aho.build\_automat()

print("\nСтруктура бора после построения автомата:")

aho.print\_trie()

suffix\_chain\_length = aho.get\_longest\_suffix\_chain()

terminal\_chain\_length = aho.get\_longest\_terminal\_chain()

print(f"\nДлина самой длинной цепочки суффиксных ссылок: {suffix\_chain\_length}")

print(f"Длина самой длинной цепочки терминальных ссылок: {terminal\_chain\_length}")

print("\nРезультаты поиска:")

matches = aho.search(text)

for position, pattern\_id in matches:

print(f"Шаблон {pattern\_id} найден на позиции {position}")

def aho\_wildcard\_test():

text = "ACTANCA"

pattern = "A$$A$"

wildcard = "$"

print(f"Текст: {text}")

print(f"Шаблон: {pattern}")

print(f"Джокер: {wildcard}")

positions = search\_with\_wildcard(text, pattern, wildcard, debug=True)

print("\nРезультаты поиска:")

for pos in positions:

print(pos)

def aho\_default(debug=False):

text = input().strip()

n = int(input().strip())

patterns = []

for \_ in range(n):

pattern = input().strip()

patterns.append(pattern)

aho = AhoCorasick()

for i, pattern in enumerate(patterns, 1):

aho.add\_pattern(pattern, i)

if debug:

print("\nСтруктура бора до построения автомата:")

aho.print\_trie()

aho.build\_automat()

if debug:

print("\nСтруктура бора после построения автомата:")

aho.print\_trie()

suffix\_chain\_length = aho.get\_longest\_suffix\_chain()

terminal\_chain\_length = aho.get\_longest\_terminal\_chain()

if debug:

print(f"Длина самой длинной цепочки суффиксных ссылок: {suffix\_chain\_length}")

print(f"Длина самой длинной цепочки терминальных ссылок: {terminal\_chain\_length}")

matches = aho.search(text)

for position, pattern\_id in matches:

print(position, pattern\_id)

if debug:

print(f"Шаблон {pattern\_id} найден на позиции {position}")

def aho\_wildcard(debug=False):

text = input().strip()

pattern = input().strip()

wildcard = input().strip()

positions = search\_with\_wildcard(text, pattern, wildcard, debug)

for pos in positions:

print(pos)

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

debug = False

# aho\_wildcard\_test()

# aho\_default\_test()

# aho\_wildcard(debug)

aho\_default(debug)