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. Вариант 2.

Студентка гр. 3343 _____ Ермолаева В. А.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Реализовать алгоритм Ахо-Корасика для поиска набора подстрок в строке. Решить задачу точного поиска для одного образца с джокером.

Задание

1) Разработайте программу, решающую задачу точного поиска набора образцов.

Вход: Первая строка содержит текст (T, $1 \leq |T| \leq 100000$). Вторая - число n ($1 \leq n \leq 3000$), каждая следующая из nn строк содержит шаблон из набора $P=\{p_1,\ldots,p_n\}$ $1 \leq |p_i| \leq 75$. Все строки содержат символы из алфавита {A,C,G,T,N}.

Выход: Все вхождения образцов из P в T. Каждое вхождение образца в текст представить в виде двух чисел - i p. Где i - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p (нумерация образцов начинается с 1). Строки выхода должны быть отсортированы по возрастанию, сначала номера позиции, затем номера шаблона.

2) Используя реализацию точного множественного поиска, решите задачу точного поиска для одного образца с джокером. В шаблоне встречается специальный символ, именуемый джокером (wild card), который "совпадает" с любым символом. По заданному содержащему шаблоны образцу P необходимо найти все вхождения P в текст T. Например, образец ab??c? с джокером ? встречается дважды в тексте 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, $1 \leq |T| \leq 100000$)

Шаблон (P, $1 \leq |P| \leq 40$)

Символ джокера

Выход: Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер). Номера должны выводиться в порядке возрастания.

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

Выполнение работы

Алгоритм Ахо-Корасика решает задачу поиска для каждой строки все ее вхождения в текст. Реализуется путем построения бора из строк за время O(m), где m - суммарная длина строк. Бор - это структура данных, которая устроена в виде дерева, с ребрами, подписанными символами, и вершинами, некоторые из которых помечены терминальными.

После построения бор преобразуется, к нему добавляются суффиксные и терминальные ссылки. Суффиксная ссылка для каждой вершины — это такая вершина, в которой оканчивается самый длинный суффикс строки, соответствующей этой вершине. Суффиксная ссылка корня бора ведет в саму себя. Терминальные же, или сжатые суффиксные, ссылки - это ссылка на ближайшую вершину, помеченную терминальной.

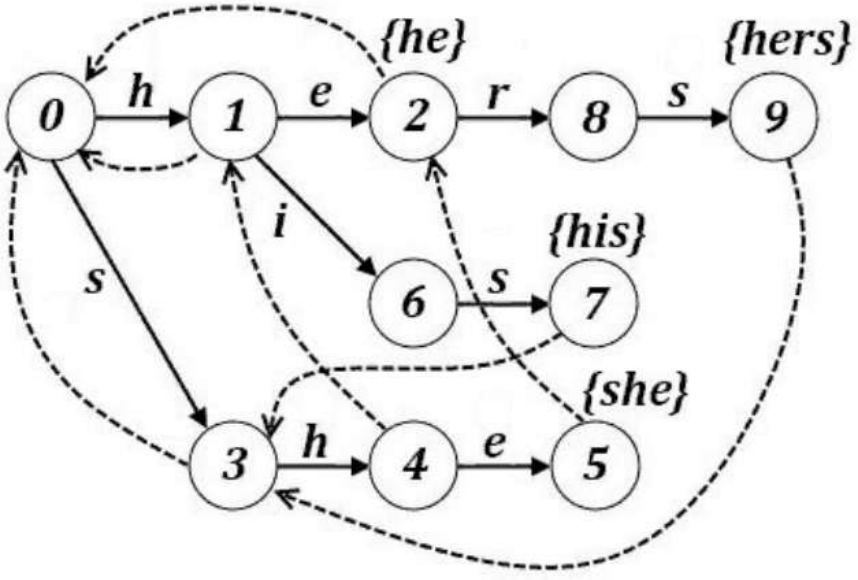

Рис. 1 - Пример автомата Ахо-Корасик

Автомат используется путем поочередного просматривания символов текста. Для очередного символа осуществляется переход из текущего состояния в состояние, которое вернёт автоматный переход. Если в новом состоянии находится терминальная вершина, то подстрока найдена.

Помимо этого был реализован алгоритм поиска шаблонов с масками. Для этого он разбивается на непрерывные подстроки без масок, которые добавляются в автомат. Далее автомат используется для поиска всех их вхождений в текст. Когда подстрока Qi находится в тексте на позиции j, это означает, что шаблон может начинаться в позиции j - li + 1 (где li - позиция начала Qi в шаблоне). Для этого заводится массив счётчиков C, где C[i] - количество подстрок, начавшихся в тексте в такой позиции, что шаблон может начинаться с i. Если в какой-либо позиции i счётчик равен числу подстрок (k), значит, в позиции i найдено полное вхождение шаблона с масками.

Временная сложность алгоритма Ахо-Корасика составляет O(n + m + a), где n - длина шаблона, m - длина текста, a - количество найденных подстрок. Пространственная же сложность составляет O(n * m) из-за необходимости хранить максимальное чисто потомков узла.

Описание реализованных классов:

- Node:
  - __init__(self, alphabet_size): инициализирует вершину автомата и имеет следующие поля:
    - self.next = [None] * alphabet_size - массив сыновей
    - self.jump = [None] * alphabet_size - массив переходов
    - self.parent = None - вершина родитель
    - self.suff_link = None - суффиксная ссылка
    - self.term_link = None - терминальная ссылка
    - self.is_leaf = False - является ли вершина терминальной
    - self.char = '' - символ вершины

        - self.leaf_pattern_number = [] - номера строк, за которые отвечает терминальная вершина
        - self.pattern = '' - шаблон
- AhoCorasick:
    - __init__(self): инициализирует автомат и имеет следующие поля:
        - self.alphabet = ['A', 'C', 'G', 'T', 'N'] - алфавит
        - self.alphabet_size = len(self.alphabet) - размер алфавита
        - self.root = Node(self.alphabet_size) - корень автомата
        - self.root.parent = self.root - родитель корня - сам корень
        - self.root.suff_link = self.root - суффиксная ссылка корня - сам корень
        - self.root.term_link = self.root - терминальная ссылка корня - сам корень
        - self.patterns = [] - шаблоны
        - self.node_count = 1 - количество вершин в автомате
    - get_char(self, v): возвращает символ вершины или 'root', если вершина – корень.
    - get_suff(v) – строит строку, соответствующую пути от корня до вершины v (т.е. суффикс).
    - jump(v, c_index) – делает переход по символу c_index с учетом суффиксных ссылок (реализует автомата-движение).
    - get_suff_link(v) – находит или возвращает уже построенную суффиксную ссылку для вершины v.
    - get_term_link(v) – находит ближайшую терминальную вершину по суффиксным ссылкам (терминальную ссылку).
    - add_string(s, pattern_number) – добавляет шаблон s в бор, помечая конечную вершину как терминальную.
    - process_text(text) – ищет все шаблоны (ранее добавленные в бор) в тексте.

- process_text_with_mask(pattern, text, wildcard) – реализует поиск шаблона с масками: разбивает по подстрокам, строит бор, считает вхождения, сверяет по позиции.

Исходный код программы смотреть в приложении А.

Тестирование

Результаты тестирования представлены в таблице 1.

Табл. 1. – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	text = "NCTAGTCN" patterns = ['TAGT', 'TCN', 'TA', 'TG', 'GT', 'NC']	1 6 3 1 3 3 5 5 6 2	Результат соответствует ожиданиям.
2.	text = "AANCTGATAAACANA" wildcard_pattern = "AA?C??A" wildcard = '?'	1 9	Результат соответствует ожиданиям.
3.	text = "AANCTGATAAACANA" wildcard_pattern = "?" wildcard = '?'	Шаблон состоит только из масок.	Результат соответствует ожиданиям.

Выводы

В ходе выполнения лабораторной работы был реализован алгоритм Ахо-Корасика для поиска набора подстрок в строке. Также была решена задача точного поиска для одного образца с джокером.

ПРИЛОЖЕНИЕ А

```
class Node:
    def __init__(self, alphabet_size):
        self.next = [None] * alphabet_size
        self.jump = [None] * alphabet_size
        self.parent = None
        self.suff_link = None
        self.term_link = None
        self.is_leaf = False
        self.char = ''
        self.leaf_pattern_number = []
        self.pattern = ''

class AhoCorasick:
    def __init__(self):
        self.alphabet = ['A', 'C', 'G', 'T', 'N']
        self.alphabet_size = len(self.alphabet)
        self.root = Node(self.alphabet_size)
        self.root.parent = self.root
        self.root.suff_link = self.root
        self.root.term_link = self.root
        self.patterns = []
        self.node_count = 1

    def get_char(self, v):
        return f"{v.char if v.char != '' else 'root'}"

    def get_suff(self, v):
        if v.char == '':
            return 'root'
        result = v.char
        while v.parent is not self.root:
            result += v.parent.char
            v = v.parent
        return result[::-1]

    def jump(self, v, c_index):
        if v.jump[c_index] is None:
            if v.next[c_index] is not None:
                v.jump[c_index] = v.next[c_index]
                print(f"\t -> Перейдем по бору {self.get_suff(v)} --->
{self.get_char(v.jump[c_index])}.")
            elif v == self.root:
                print(f"\t -> Перешли в root.")
                v.jump[c_index] = self.root
            else:
                v.jump[c_index] = self.jump(self.get_suff_link(v),
c_index)
                # print(f"\t -> Перейдем по суффиксной ссылке
{self.get_suff(v)} ---> {self.get_suff(v.jump[c_index])}.")
        else: print(f"\t -> Переход по автомату {self.get_suff(v)} --->
{self.get_suff(v.jump[c_index])}.")
        return v.jump[c_index]

    def get_suff_link(self, v):
        if v.suff_link is None:
            if v == self.root or v.parent == self.root:
                v.suff_link = self.root
            else:
                print(f"\tИщем суффикс в боре:")
                c_index = self.alphabet.index(v.parent.char)
```

```
                    v.suff_link = self.jump(self.get_suff_link(v.parent),
c_index)
                    if v.suff_link != self.root: print("\tСуффикс найден.")
                    else: print("\tМаксимальный суффикс пустой.")
                print(f"\t -> Строим суффиксную ссылку {self.get_suff(v)} --->
{self.get_suff(v.suff_link)}")
            else: print(f"\t -> Переходим по суффиксной ссылке
{self.get_suff(v)} ---> {self.get_suff(v.suff_link)}.")
            return v.suff_link

        def get_term_link(self, v):
            if v.term_link is None:
                suff_link = self.get_suff_link(v)
                if suff_link.is_leaf:
                    v.term_link = suff_link
                elif suff_link == self.root:
                    v.term_link = self.root
                else:
                    v.term_link = self.get_term_link(suff_link)
                print(f"\t -> Строим терминальную ссылку {self.get_suff(v)}
---> {self.get_suff(v.term_link)}.")
            else: print(f"\t -> Переходим по терминальной ссылке
{self.get_suff(v)} ---> {self.get_suff(v.term_link)}.")
            return v.term_link

        def add_string(self, s, pattern_number):
            print(f"Добавим строку '{s}' в бор:")
            cur = self.root
            for c in s:
                c_index = self.alphabet.index(c)
                if cur.next[c_index] is None:
                    print(f"\t -> Символ '{c}' отсутствует в боре, добавляем
его.")
                    new_node = Node(self.alphabet_size)
                    new_node.char = c
                    new_node.parent = cur
                    new_node.parent.char = c
                    cur.next[c_index] = new_node
                    self.node_count += 1
                else: print(f"\t -> Символ '{c}' уже существует в боре.")
                cur = cur.next[c_index]
            print(f"\t -> Помечаем символ '{c}' терминальным.")
            cur.is_leaf = True
            cur.leaf_pattern_number.append(pattern_number)
            cur.pattern = s
            self.patterns.append(s)

        def process_text(self, text):
            result = []
            current = self.root

            for i, c in enumerate(text):
                print(f"Рассмотрим вершину {c} на позиции {i + 1} в тексте
{text}:")
                c_index = self.alphabet.index(c)
                current = self.jump(current, c_index)
                if current.char == '': print("\t -> Подстрока не встречается в
тексте.")
                else: print(f"\tПерешли в состояние
{self.get_suff(current)}.")

                temp = current
                while temp != self.root:
```

```
                    term = self.get_term_link(temp)
                    if temp.is_leaf:
                        for pattern_num in temp.leaf_pattern_number:
                            pattern_length = len(self.patterns[pattern_num])
                            start_pos = i - pattern_length + 1
                            result.append((start_pos, pattern_num))
                        print(f"\t -> Вершина {c} терминальная, обнаружено
вхождение подстроки {temp.pattern}.")
                        print(f"\t -> Переходим по терминальной ссылке
{self.get_suff(temp)} ---> {self.get_suff(term)}.")
                    temp = term

            print(f"Количество вершин в автомате = {self.node_count}.")
            return result

        def process_text_with_mask(self, pattern, text, wildcard):
            print(f"----------------\nШаг 0. Проверка, что шаблон не состоит
только из масок.")
            if all(ch == wildcard for ch in pattern):
                print(" -> Шаблон состоит только из масок.")
                return []

            print("Шаг 1. Разобьем строку на подстроки без маскок.")
            substrings = []
            substring_positions = []
            i = 0
            while i < len(pattern):
                if pattern[i] == wildcard:
                    i += 1
                    continue
                start = i
                while i < len(pattern) and pattern[i] != wildcard:
                    i += 1
                substrings.append(pattern[start:i])
                substring_positions.append(start)
            print(f"Подстроки без масок: {", ".join(substrings)} на позициях:
{", ".join(map(str, substring_positions))}.")

            print("----------------\nШаг 2. Добавим подстроки в бор.")
            for i, substring in enumerate(substrings):
                self.add_string(substring, i)

            counter = [0] * len(text)
            current = self.root
            print(f"----------------\nШаг 3. Инициализируем счетчик
совпадений: {counter}")

            print(f"----------------\nШаг 4. Подсчитаем вхождения подстрок.")
            for i, c in enumerate(text):
                print(f"Рассмотрим вершину {c} на позиции {i + 1} в тексте
{text}:")
                c_index = self.alphabet.index(c)
                current = self.jump(current, c_index)
                if current.char == '': print("\t -> Подстрока не встречается в
тексте.")
                else: print(f"\tПерешли в состояние
{self.get_suff(current)}.")

                temp = current
                while temp != self.root:
                    term = self.get_term_link(temp)
                    if temp.is_leaf:
                        for pattern_num in temp.leaf_pattern_number:
```

```
                                substring_position =
substring_positions[pattern_num]
                                substring_length = len(substrings[pattern_num])
                                start_pos = i - substring_length -
substring_position + 1
                                counter[start_pos] += 1
                            print(f"\t -> Вершина {temp.char} терминальная,
обнаружено вхождение подстроки {temp.pattern}.")
                        temp = term

            print(f"----------------\nШаг 5. Найдем вхождения шаблона.")
            print(f"Получившийся счетчик совпадений: {counter}.")
            result = []
            for i, count in enumerate(counter):
                if count == len(substrings):
                    result.append(i + 1)
                    print(f"\t -> Количество вхождений совпало для позиции {i
+ 1} с числом {count}.")
            return result

    # var = int(input("Выберите вариант:\n\t1. Поиск набора образцов.\n\t2.
Поиск образца с джокером.\n"))
    var = 2

    # Задание 1
    if var == 1:
        text = input().strip()
        n = int(input())
        patterns = [input() for _ in range(n)]

        ac = AhoCorasick()
        print("Шаг 1. Добавим все строки в бор.")
        for i, pattern in enumerate(patterns):
            ac.add_string(pattern, i)

        print("----------------\nШаг 2. Преобразуем бор.")
        matches = ac.process_text(text)

        print("----------------\nШаг. 3. Вывод вхождений в текст.")
        matches.sort()
        for pos, pattern_num in matches:
            print(f" -> Шаблон {patterns[pattern_num]} встречается в тексте
{text} на позиции {pos}.")

        print("----------------\nШаг. 4. Вывод найденных пересечений.")
        positions = [-1 for _ in range(len(text))]
        for pos, pattern_num in matches:
            length = len(patterns[pattern_num])
            for i in range(length):
                if positions[pos + i] != -1:
                    print(f" -> Шаблон {patterns[pattern_num]} пересекается с
шаблоном {patterns[positions[pos + i]]}.")
                    break
                positions[pos + i] = pattern_num

    # Задание 2
    else:
        text = input()
        wildcard_pattern = input()
        wildcard = input()

        ac = AhoCorasick()
        matches = ac.process_text_with_mask(wildcard_pattern, text, wildcard)
```

```
        print(f"----------------\nШаг 6. Вывод найденных вхождений шаблона.")
        print(f" -> Шаблон {wildcard_pattern} встречается в тексте {text} на
позициях: {", ".join(map(str, matches))}.")
```