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,$1 \leq |T| \leq 100000$T).

Вторая - число nn ($1 \leq n \leq 3000$), каждая следующая из nn строк содержит шаблон из набора $P=\{p1,\ldots,pn\}$ $1 \leq |pi| \leq 75$

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 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 P в текст T.

Например, образец ab??c?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≤ | 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)
```