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 _____ Снигирев А.А

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы

Изучить алгоритм Ахо-Корасик. Реализовать его классическое применение поиска нескольких подстрок в строке и задачу поиска шаблона по маске.

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 n 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.

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 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, $1 \le |T| \le 100000$)

Шаблон (P, $1 \le |P| \le 40$)

Символ джокера

Выход:

Строки с номерами позиций вхождений шаблона (каждая строка содержит только один номер).

Номера должны выводиться в порядке возрастания.

Выполнение работы

Архитектура алгоритма строится в несколько этапов:

- Построение бора паттернов.
- Модификация бора суффиксными ссылками до автомата.
- Алгоритм поиска подстрок в тексте, использующий созданный автомат.

Часть 1. Бор

Бор строится довольно просто. Каждое ребро дерева означает символ. Символы строки выбираются параллельно с переходами по бору. Если из текущей вершины есть переход по символу строки, то выполняется переход, иначе создается новая вершина, а получившееся ребро именуется символом.

Когда строка заканчивается, конечная вершина помечается как *терминальная,* т.е символизирующая, что здесь кончается один из паттернов.

За построение бора отвечает первая часть функции build_automaton()

Часть 2. Суффиксные ссылки

Суффиксная ссылка для вершины А это дополнительное ребро, ведущее в вершину Б, префикс которой является максимальным в боре суффиксом А.

Пусть мы скормили бору какую-нибудь подстроку s1, равную одному из паттернов, и оказались в соответствующей терминальной вершине. Предположим, в наборе паттернов есть строка s2, некоторый префикс которой совпадает с суффиксом строки s1, тогда между вершинами есть суфссылка, перейдя по которой алгоритм попытает счастье еще и со строкой s2. Суффиксные ссылки позволяют алгоритму совершать быстрые переходы между ветвями бора, что дает возможность найти сразу несколько подстрок за один проход по тексту.

У детей root такие ссылки ведут в сам root. Также очевидно, что все такие ссылки переходят на более верхние уровни дерева.

Используется обход в ширину. Для каждой новой вершины проверяется суффиксная ссылка ее родителя. Происходит переход по ссылкам, пока не

будет достигнут корень или не будет найден переход по символу. Бор с суффиксными ссылками является полным автоматом.

«Хорошие» ссылки позволяю не перемещаться долго по суффиксным ссылкам, а сразу хранят индекс вершины, которая является терминальной. Это позволяет быстро собрать все совпадения, не совершая рекурсивных обходов суффиксных путей.

За построение полного автомата строк в реализации отвечает метод build_automaton()

Часть 3. Поиск подстрок

Во время поиска в тексте алгоритм переходит по состояниям автомата, следуя символам текста. Когда он достигает состояния current_state, он собирает все совпадения шаблонов следующим образом: Проверяет output[current_state]. Если список не пуст, добавляет соответствующие шаблоны в результаты (с позицией, вычисленной как i - len(pattern) + 1, где i — текущая позиция в тексте). Затем переходит по output_link[current_state] к следующему состоянию с непустым output (если output_link != -1) и добавляет шаблоны из output этого состояния.

Этот процесс повторяется, пока не будет достигнуто состояние с output_link = 1 или состояние не было посещено ранее (чтобы избежать зацикливания).

Задание 2:

Для нахождения всех совпадений с шаблоном-маской используется довольно интересный алгоритм.

Шаблон разбивается на бездджокерные строки с помощью токенизации по символу-джокеру. При этом запоминаются индексы вхождения li в шаблон для каждой подстроки. Объявляется массив C.

Далее применяется алгоритм Ахо-Корасик для поиска всех токенов. При нахождении токена вычисляется индекс j-li+1, где j – позиция вхождения. Элемент массива C по этому индексу инкрементируется.

После нахождения всех токенов нужно просмотреть, на каких позициях в массиве C расположены числа ==k, где k – число подстрок в шаблоне, утверждается, что это и есть вхождения маски.

Теоретическая оценка сложности:

Задание 1:

Построение Бора — O(s)

s – суммарная длина подстрок

Проход по тексту — O(n)

n – длина текста

Сбор совпадений — O(z)

Память:

O(s) – хранение Бора

O(n) – хранение очереди, суффиксных и прямых ссылок

Итого O(s+n)

Исходный код см. в **ПРИЛОЖЕНИИ А**

Вывод: Алгоритм Ахо-Корасик крайне эффективен в задаче множественного поиска и связанных с ней. Ищет подстроки за линейное время. Однако для требуются некоторые ресурсы для построения автомата.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
from collections import deque
DEBUG = True

def find_pattern_with_wildcards(T, P, wildcard):
    if DEBUG:
        print("=== Starting find_pattern_with_wildcards ===")
    if DEBUG:
        print(f"Text: '{T}'")
    if DEBUG:
        print(f"Pattern: '{P}'")
    if DEBUG:
        print(f"Wildcard: '{wildcard}'")

    # Разбиение шаблона на подстроки
    if DEBUG:
        print("\nSplitting pattern by wildcard:")
    substrings = [s for s in P.split(wildcard) if s]
    if DEBUG:
        print(f"  Substrings: {substrings}")
    if not substrings:
        if DEBUG:
            print("  No non-empty substrings found, returning
empty result")
        if DEBUG:
            print("=== find_pattern_with_wildcards Completed
===")
        return []

    k = len(substrings)
    if DEBUG:
        print(f"  Number of substrings (k): {k}")

    # Вычисление стартовых позиций
    if DEBUG:
```

```
            print("\nComputing start positions of substrings in
pattern:")
        start_positions = []
        pos = 0
        for i, sub in enumerate(P.split(wildcard)):
            if sub:
                start_positions.append(pos)
                if DEBUG:
                    print(f"  Substring '{sub}' starts at position
{pos}")
            pos += len(sub) + (1 if i < len(P.split(wildcard)) - 1
else 0)
        if DEBUG:
            print(f"  Start positions: {start_positions}")

        # Создание автомата и поиск
        if DEBUG:
            print("\nBuilding Aho-Corasick automaton and
searching:")
        ac = AhoCorasick(substrings)
        matches = ac.search(T)
        if DEBUG:
            print(f"  Matches found: {matches}")
        for pos, pattern_idx in matches:
            if DEBUG:
                print(f"    Pattern {pattern_idx}
('{substrings[pattern_idx]}') at position {pos}")

        # Подсчет совпадений
        if DEBUG:
            print("\nCounting potential matches:")
        n = len(T)
        C = [0] * n
        for pos, pattern_idx in matches:
            start_pos_in_P = start_positions[pattern_idx]
            text_start = pos - start_pos_in_P
            if text_start >= 0:
                C[text_start] += 1
                if DEBUG:
```

```
                    print(f"  Match for pattern {pattern_idx} at
pos {pos} -> text_start = {text_start}, C[{text_start}] =
{C[text_start]}")
            else:
                if DEBUG:
                    print(f"  Match for pattern {pattern_idx} at
pos {pos} -> text_start = {text_start}, skipped (negative)")
        if DEBUG:
            print(f"  Count array C: {C}")

        # Проверка полных вхождений
        if DEBUG:
            print("\nChecking for complete pattern matches:")
        result = []
        for i in range(n):
            if C[i] == k and i + len(P) - 1 < n:
                if DEBUG:
                    print(f"  Position {i}: C[{i}] = {k}, checking
if valid match")
                valid = True
                for j in range(len(P)):
                    text_pos = i + j
                    if P[j] != wildcard and T[text_pos] != P[j]:
                        if DEBUG:
                            print(f"    Mismatch at pattern pos
{j}: P[{j}] = '{P[j]}', T[{text_pos}] = '{T[text_pos]}'")
                        valid = False
                        break
                    else:
                        status = "wildcard" if P[j] == wildcard
else f"matches '{T[text_pos]}'"
                        if DEBUG:
                            print(f"    Pattern pos {j}: P[{j}] =
'{P[j]}', T[{text_pos}] = '{T[text_pos]}' ({status})")
                if valid:
                    result.append(i + 1)
                    if DEBUG:
                        print(f"    Valid match found at position
{i + 1} (1-based)")
```

```
                else:
                    if DEBUG:
                        print(f"    Invalid match at position {i}")
            else:
                if C[i] != k:
                    if DEBUG:
                        print(f"  Position {i}: C[{i}] != {k},
skipping")
                else:
                    if DEBUG:
                        print(f"  Position {i}: Match would exceed
text length, skipping")
        if DEBUG:
            print(f"\nFinal result: {result}")
        if DEBUG:
            print("=== find_pattern_with_wildcards Completed ===")
        return sorted(result)

    class AhoCorasick:
        def __init__(self, patterns):
            self.patterns = [p for p in patterns if p]
            self.num_patterns = len(self.patterns)
            self.max_states = sum(len(p) for p in self.patterns) +
1
            self.transitions = [{} for _ in range(self.max_states)]
            self.output = [[] for _ in range(self.max_states)]
            self.fail = [0] * self.max_states
            self.output_link = [-1] * self.max_states
            self.state_counter = 1
            self.build_automaton()

        def build_automaton(self):
            if DEBUG:
                print("=== Building Aho-Corasick Automaton ===")
            root = 0
            self.fail[root] = root
            self.output_link[root] = -1
            self.state_counter = 1
            if DEBUG:
```

```
                print(f"Initialized root state {root}: fail[{root}]
= {root}, output_link[{root}] = -1")
            if DEBUG:
                print("\nBuilding Trie:")
            for i, pattern in enumerate(self.patterns):
                if DEBUG:
                    print(f"  Adding pattern {i}: '{pattern}'")
                current_state = root
                for char in pattern:
                    if char not in self.transitions[current_state]:
                        if DEBUG:
                            print(f"    Created new state
{self.state_counter} for char '{char}' from state {current_state}")
                        self.transitions[current_state][char] =
self.state_counter
                        self.state_counter += 1
                    else:
                        if DEBUG:
                            print(f"    Using existing state
{self.transitions[current_state][char]} for char '{char}' from state
{current_state}")
                    current_state = self.transitions[current_state]
[char]
                self.output[current_state].append(i)
                if DEBUG:
                    print(f"    Marked state {current_state} as end
of pattern {i}: output[{current_state}] =
{self.output[current_state]}")

            if DEBUG:
                print("\nComputing Failure and Output Links:")
            queue = deque()
            for char in self.transitions[root]:
                state = self.transitions[root][char]
                self.fail[state] = root
                self.output_link[state] = state if
self.output[state] else -1
                queue.append(state)
                if DEBUG:
```

```
                    print(f"  Root child state {state} (char
'{char}'): fail[{state}] = {root}, output_link[{state}] =
{self.output_link[state]}")

            while queue:
                current_state = queue.popleft()
                if DEBUG:
                    print(f"  Processing state {current_state}")
                for char in self.transitions[current_state]:
                    next_state = self.transitions[current_state]
[char]
                    queue.append(next_state)
                    if DEBUG:
                        print(f"    Transition from {current_state}
on '{char}' to {next_state}")

                    fail_state = self.fail[current_state]
                    while fail_state != root and char not in
self.transitions[fail_state]:
                        if DEBUG:
                            print(f"      No transition for
'{char}' in fail_state {fail_state}, moving to fail[{fail_state}] =
{self.fail[fail_state]}")
                        fail_state = self.fail[fail_state]
                    self.fail[next_state] =
self.transitions[fail_state].get(char, root)
                    if DEBUG:
                        print(f"      Set fail[{next_state}] =
{self.fail[next_state]} (found transition for '{char}' or root)")

                    fail = self.fail[next_state]
                    self.output_link[next_state] = fail if
self.output[fail] else self.output_link[fail]
                    if DEBUG:
                        print(f"      Set output_link[{next_state}]
= {self.output_link[next_state]} (based on output[{fail}] =
{self.output[fail]})")
            if DEBUG:
                print("=== Automaton Built ===")
```

```
        def search(self, text):
            if DEBUG:
                print("\n=== Searching in Text ===")
            if DEBUG:
                print(f"Text: '{text}'")
            current_state = 0
            results = []
            for i in range(len(text)):
                char = text[i]
                if DEBUG:
                    print(f"\nPosition {i}: char = '{char}',
current_state = {current_state}")
                while current_state != 0 and char not in
self.transitions[current_state]:
                    if DEBUG:
                        print(f"  No transition for '{char}' in
state {current_state}, moving to fail[{current_state}] =
{self.fail[current_state]}")
                    current_state = self.fail[current_state]

                if char in self.transitions[current_state]:
                    current_state = self.transitions[current_state]
[char]
                    if DEBUG:
                        print(f"  Transition to state
{current_state} on '{char}'")
                else:
                    current_state = 0
                    if DEBUG:
                        print(f"  No transition for '{char}', reset
to root state {current_state}")

                temp_state = current_state
                visited = set()
                if DEBUG:
                    print(f"  Collecting outputs from state
{temp_state}")
```

```
                while temp_state != -1 and temp_state not in
visited:
                    visited.add(temp_state)
                    for pattern_index in self.output[temp_state]:
                        pos = i - len(self.patterns[pattern_index])
+ 1
                        results.append((pos, pattern_index))
                        if DEBUG:
                            print(f"      Found pattern
{pattern_index} ('{self.patterns[pattern_index]}') at position
{pos}")
                    temp_state = self.output_link[temp_state]
                    if DEBUG:
                        print(f"    Moving to
output_link[{temp_state if temp_state != -1 else 'none'}] =
{temp_state}")
                if DEBUG:
                    print("=== Search Completed ===")
            return results

        def draw(self):
            dot = ["digraph AhoCorasick {"]
            dot.append("  rankdir=LR;")
            dot.append("  node [shape=circle, style=filled,
fillcolor=lightgrey];")
            dot.append("  edge [color=black];")

            for state in range(self.state_counter):
                label = f"{state}"
                if self.output[state]:
                    label += f"\\n{self.output[state]}"
                    dot.append(f"  {state} [shape=doublecircle,
fillcolor=lightgreen, label=\"{label}\"];")
                elif state == 0:
                    dot.append(f"  {state} [shape=circle,
fillcolor=lightblue, label=\"{label}\"];")
                else:
                    dot.append(f"  {state} [shape=circle,
label=\"{label}\"];")
```

```
            for state in range(self.state_counter):
                for char, next_state in
self.transitions[state].items():
                    dot.append(f"  {state} -> {next_state}
[label=\"{char}\", color=black];")

            for state in range(1, self.state_counter):
                if self.fail[state] != state:
                    dot.append(f"  {state} -> {self.fail[state]}
[style=dashed, color=red, label=\"fail\"];")

            for state in range(self.state_counter):
                if self.output_link[state] != -1 and
self.output_link[state] != state:
                    dot.append(f"  {state} ->
{self.output_link[state]} [style=dotted, color=blue,
label=\"out\"];")

            dot.append("}")
            return "\n".join(dot)

    if __name__ == "__main__":
        print("Choose function: std/joker")
        ans = input()
        if ans=="std":
            print("Enter your text: ")
            T = input().strip()
            print("Enter num of patterns: ")
            n = int(input())
            print("Enter patterns: ")
            patterns = [input().strip() for _ in range(n)]

            ac = AhoCorasick(patterns)
            result = ac.search(T)

            if DEBUG:
                print("Results: ")
            for pos, pattern_idx in result:
```

```
                    if DEBUG:
                        print(f"{patterns[pattern_idx]}: {pos}")

                dot_graph = ac.draw()
                if DEBUG:
                    print("\nDOT representation of the automaton:")
                if DEBUG:
                    print(dot_graph)

                with open("automaton.dot", "w") as f:
                    f.write(dot_graph)
            else:
                print("Enter your text: ")
                T = input().strip()
                print("Enter your mask: ")
                P = input().strip()
                print("Enter joker-symbol: ")
                wildcard = input().strip()

                occurrences = find_pattern_with_wildcards(T, P,
wildcard)
                if occurrences:
                    for pos in occurrences:
                        print(pos)
```