**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)

Вариант: 2

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3388 |  | Титкова С.Д. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

**Цель работы:**

Изучить принцип работы алгоритма Ахо-Корасик для нахождения подстрок в строке. Решить с его помощью задачи.

**Задание 1:**

Разработайте программу, решающую задачу точного поиска набора образцов.  
**Вход:**  
Первая строка содержит текст (T,1≤∣T∣≤100000T).

Вторая - число *n* (1≤n≤3000), каждая следующая из *n* строк содержит шаблон из набора *P*={*p\_*1​,…,*p\_n*​} 1≤∣*p\_i*​∣≤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 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∣≤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

**Реализация**

*Описание алгоритма Ахо-Корасика для точного поиска образцов:*

Алгоритм Ахо-Корасика предназначен для эффективного поиска всех вхождений множества заданных шаблонов P={p\_1,p\_2,…,p\_n} в текст T. Он использует структуру данных, называемую бором (trie), дополненную суффиксными и конечными ссылками, что позволяет выполнять поиск за один проход по тексту независимо от числа шаблонов. В данной реализации алгоритм также подсчитывает количество вершин в автомате и определяет шаблоны, имеющие пересечения с другими вхождениями в тексте.

**Шаги алгоритма**

Создаётся корневое состояние (0) с fail = 0 (суффиксная ссылка) и outputlink = −1 (конечная ссылка). Определяется максимальное число состояний как сумма длин всех шаблонов плюс корень (∑∣pi∣+1).

Для каждого шаблона p\_i из P. Начиная с корня, для каждого символа c в p\_i добавляется новое состояние, если перехода по c ещё нет. Счётчик вершин state\_count увеличивается на 1 для каждого нового состояния. Конечное состояние шаблона отмечается его индексом i в массиве output.

Далее строится автомат. Используется очередь для обхода состояний в порядке ширины (BFS). Для каждого состояния из корня: fail = 0, output\_link устанавливается как само состояние, если оно содержит шаблон, иначе -1.

Для остальных состояний. Суффиксная ссылка fail[next\_state] определяется поиском самого длинного суффикса текущей строки, доступного через fail[current\_state]. Конечная ссылка output\_link[next\_state] указывает на ближайший узел с шаблоном по цепочке fail, либо на −1.

Затем производится поиск в тексте. Проход по символам текста T с использованием текущего состояния current\_state. Если перехода по символу нет, алгоритм следует по fail до корня или подходящего состояния. При достижении состояния с непустым output или через output\_link фиксируются все вхождения шаблонов с вычислением их позиций.

Также по заданию из варианта требовалось осуществить: подсчёт вершин: возвращается state\_count. Определение пересечений: для каждого вхождения вычисляются начало и конец, проверяются пересечения между парами вхождений.

.

*Описание функций и структур:*

Был создан класс *AhoCorasick* в нём определены следующие поля:

* *Patterns* - используется для построения бора и вычисления позиций вхождений.
* *Transitions* - представляет бор, обеспечивая быстрый доступ к следующему состоянию
* *Output* - хранит информацию о завершении шаблонов в вершинах бора
* *Fail* - суффиксные ссылки для быстрого перехода при несовпадении символов
* *Output\_link* - конечные ссылки для эффективного извлечения всех совпадени
* *State\_count* - используется для подсчёта вершин и ограничения массивов.

Также были реализованы методы класса:

* *\_\_init\_\_(self, patterns)* – метод, который инициализирует автомат Ахо-Корасика для заданного набора шаблонов.
* *Build\_automaton(self)* – метод, который строит бор и автомат Ахо-Корасика с суффиксными и конечными ссылками.
* *Search(self, text)* - метод, который ищет все вхождения шаблонов в тексте
* *Get\_vertex\_count(self)* – метод, который возвращает количество вершин в автомате
* *Find\_overlapping\_patterns(self, text, results)* – метод, который определяет шаблоны, вхождения которых пересекаются с другими.

Также была написана функция *process\_search(T, patterns)*, которая обрабатывает поиск шаблонов в тексте и возвращает результаты.

*Оценка сложности алгоритма:*

**Временная сложность**

Построение бора:

* + Для каждого символа всех шаблонов создаётся или используется состояние: O(∑∣pi∣), где| ∑∣pi​∣ — суммарная длина шаблонов

Построение автомата:

* + Обход BFS по всем состояниям (до ∑∣pi∣+1) с проверкой переходов для каждого символа алфавита.
  + O(∑∣pi​∣) для небольшого алфавита, так как ∣Σ∣ считается константой.

Поиск:

* + Проход по тексту: O(∣T∣) переходов, каждый с O(1) по fail и output\_link благодаря конечным ссылкам.
  + Извлечение совпадений: O(z), где z — число вхождений.

*Итог:* O(∣T∣+z)

**Пространственная сложность**

Бор:

* + transitions: O(∑∣pi∣) состояний, каждый с O(∣Σ∣) переходов (в худшем случае).
  + output: O(∑∣pi∣) для хранения индексов шаблонов.

Ссылки:

* + fail, output\_link: O(∑∣pi∣) элементов.

Результаты:

* + results: O(z) пар.
  + overlap\_patterns: O(n) в худшем случае, где n — число шаблонов.

*Итог:* O(∑∣pi∣) для автомата плюс O(z) для результатов, итого O(∑∣pi∣+z).

**Тестирование**

Таблица 1. Тестирование.

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| ACGT  2  ACGTACGT  CGTA |  |
| AAAAA  2  A  AA | 1 1  1 2  2 1  2 2  3 1  3 2  4 1  4 2  5 1 |
| ACGTACGT  3  AC  CG  GT | 1 1  2 2  3 3  5 1  6 2  7 3 |
| ACGTACGT  3  A  AC  ACG | 1 1  1 2  1 3  5 1  5 2  5 3 |

*Описание Алгоритма Ахо-Корасика с джокерами:*

Алгоритм Ахо-Корасика в данной реализации адаптирован для поиска всех вхождений шаблона P с джокерами (специальный символ, обозначающий совпадение с любым символом) в тексте T. Он использует бор (trie) с суффиксными и конечными ссылками для поиска всех безджокерных подстрок шаблона P, а затем проверяет полное соответствие P в найденных позициях с учётом джокеров. Алгоритм также подсчитывает количество вершин в автомате и определяет шаблоны с пересекающимися вхождениями.

**Шаги алгоритма**

Создаётся корневое состояние (0) с fail = 0 (суффиксная ссылка) и outputlink = −1 (конечная ссылка). Определяется максимальное число состояний как сумма длин всех шаблонов плюс корень (∑∣pi∣+1).

Для каждой подстроки Q\_i из P. Начиная с корня, создаются состояния для каждого символа c, если перехода нет. Счётчик state\_count увеличивается для новых состояний. Конечное состояние подстроки помечается её индексом в output.

Далее производится построение автомата. Для этого будет использоваться обход в ширину (BFS). Для детей корня: fail = 0, output\_link— само состояние, если оно конец шаблона, иначе −1. Для остальных: fail определяется поиском через fail родителя, output\_link— ближайший узел с шаблоном.

Далее проходим по T с использованием transitions и fail для переходов. Совпадения извлекаются через output и output\_link.

Для каждого совпадения Q\_i на позиции j: C[j - l\_i] += 1, где l\_i — стартовая позиция Q\_i в P. Позиции i, где C[i] = k и T[i:i+|P|] соответствует P с джокерами, добавляются в результат.

Также по заданию из варианта требовалось осуществить: подсчёт вершин: возвращается state\_count. Определение пересечений: для каждого вхождения вычисляются начало и конец, проверяются пересечения между парами вхождений

*Описание функций и структур:*

Был создан класс *AhoCorasick* в нём определены следующие поля:

* *Patterns* - используется для построения бора и вычисления позиций вхождений.
* *Transitions* - представляет бор для поиска подстрок.
* *Output* - хранит совпадения подстрок
* *Fail* - суффиксные ссылки для быстрого перехода при несовпадении символов
* *Output\_link* - конечные ссылки для эффективного извлечения всех совпадени
* *State\_count* - используется для подсчёта вершин и ограничения массивов.

Также были реализованы методы класса:

* *\_\_init\_\_(self, patterns)* – метод, который инициализирует автомат Ахо-Корасика для заданного набора шаблонов.
* *Build\_automaton(self)* – метод, который строит бор и автомат Ахо-Корасика с суффиксными и конечными ссылками.
* *Search(self, text)* - метод, который ищет все вхождения шаблонов в тексте
* *Get\_vertex\_count(self)* – метод, который возвращает количество вершин в автомате
* *Find\_overlapping\_patterns(self, text, results)* – метод, который определяет шаблоны с пересечениями

Также была написана функция *find\_pattern\_with\_wildcards(T, P, wildcard)*, которая находит вхождения P с джокерами в T.

*Оценка сложности алгоритма:*

**Временная сложность:**

Разбиение P:

* + O(∣P∣) для разбиения по wildcard и вычисления start\_positions.

Построение бора:

* + O(∑∣Qi​∣) для добавления подстрок, где ∑∣Qi​∣≤∣P∣.

Построение автомата:

* + O(∑∣Qi∣) для BFS с фиксированным алфавитом (∣Σ∣ — константа).

Поиск:

* + O(∣T∣) для прохода по тексту, O(z) для извлечения совпадений (z — число вхождений).

Проверка джокеров:

* + O(∣T∣⋅∣P∣) для анализа C и проверки символов P в каждой позиции.

Пересечения:

* + O(z^2) для сравнения всех пар вхождений.

*Итого:* O(∑∣Qi∣+∣T∣⋅∣P∣+z2)

**Пространственная сложность**

Бор:

* + transitions: O(∑∣Qi∣) состояний.
  + output, fail, ooutput\_link: O(∑∣Qi∣).

Поиск:

* + C: O(∣T∣).
  + results: O(z) O(z) O(z).
  + overlap\_patterns: O(k), где k≤∣P∣.

*Общая:* O(∑∣Qi∣+∣T∣+z), z≤∣T∣.

**Тестирование**

Таблица 2. Тестирование.

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| AAAAA  A\*A  \* | 1  2  3 |
| ACGTACGT  \*CG\*  \* | 1  5 |
| ACTANCA  A$$A$  $ | 1 |
| NTAG  T\*G  \* | 2 |

**Вывод**

В ходе лабораторной работы были написаны программы с использованием алгоритма Ахо-Корасика. Также дополнительно было сделано: подсчёт вершин и определение пересечений.

**Исходный код программы см. в ПРИЛОЖЕНИИ А.**

**ПРИЛОЖЕНИЕ А.**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

aho\_1.py

from collections import deque  
  
DEBUG = False  
  
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\_count = 0  
 self.build\_automaton()  
  
 def build\_automaton(self):  
 root = 0  
 self.fail[root] = root  
 self.output\_link[root] = -1  
 self.state\_count = 1  
  
 if DEBUG:  
 print("Построение бора:")  
 for i in range(self.num\_patterns):  
 current\_state = root  
 for char in self.patterns[i]:  
 if char not in self.transitions[current\_state]:  
 self.transitions[current\_state][char] = self.state\_count  
 if DEBUG:  
 print(f"Добавлено ребро из состояния {current\_state} в состояние {self.state\_count} по символу '{char}'")  
 self.state\_count += 1  
 current\_state = self.transitions[current\_state][char]  
 self.output[current\_state].append(i)  
 if DEBUG:  
 print(f"Добавлен шаблон {i} в состояние {current\_state}")  
  
 if DEBUG:  
 print("\nПостроение автомата:")  
 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"Установлен fail-переход для состояния {state} в состояние {root}")  
  
 while queue:  
 current\_state = queue.popleft()  
 for char in self.transitions[current\_state]:  
 next\_state = self.transitions[current\_state][char]  
 queue.append(next\_state)  
  
 fail\_state = self.fail[current\_state]  
 while fail\_state != root and char not in self.transitions[fail\_state]:  
 fail\_state = self.fail[fail\_state]  
 self.fail[next\_state] = self.transitions[fail\_state].get(char, root)  
 if DEBUG:  
 print(f"Установлен fail-переход для состояния {next\_state} в состояние {self.fail[next\_state]}")  
  
 fail = self.fail[next\_state]  
 self.output\_link[next\_state] = fail if self.output[fail] else self.output\_link[fail]  
 if DEBUG:  
 print(f"Установлен output-переход для состояния {next\_state} в состояние {self.output\_link[next\_state]}")  
  
 if DEBUG:  
 print("\nПостроенный автомат:")  
 for state in range(self.state\_count):  
 print(f"Состояние {state}:")  
 print(f" Переходы: {self.transitions[state]}")  
 print(f" Выходы: {self.output[state]}")  
 print(f" Fail-переход: {self.fail[state]}")  
 print(f" Output-переход: {self.output\_link[state]}")  
  
 def search(self, text):  
 current\_state = 0  
 results = []  
 if DEBUG:  
 print("\nПроцесс поиска:")  
 for i in range(len(text)):  
 char = text[i]  
 while current\_state != 0 and char not in self.transitions[current\_state]:  
 current\_state = self.fail[current\_state]  
 if DEBUG:  
 print(f"Переход по fail-ссылке в состояние {current\_state}")  
  
 if char in self.transitions[current\_state]:  
 current\_state = self.transitions[current\_state][char]  
 if DEBUG:  
 print(f"Переход в состояние {current\_state} по символу '{char}'")  
 else:  
 current\_state = 0  
 if DEBUG:  
 print(f"Символ '{char}' не найден, переход в корневое состояние")  
  
 temp\_state = current\_state  
 visited = set()  
 while temp\_state != -1 and temp\_state not in visited:  
 visited.add(temp\_state)  
 if self.output[temp\_state]:  
 for pattern\_index in self.output[temp\_state]:  
 pos = i - len(self.patterns[pattern\_index]) + 2  
 results.append((pos, pattern\_index + 1))  
 if DEBUG:  
 print(f"Найден шаблон {pattern\_index + 1} на позиции {pos}")  
 temp\_state = self.output\_link[temp\_state]  
 return sorted(results)  
  
 def get\_vertex\_count(self):  
 return self.state\_count  
  
 def find\_overlapping\_patterns(self, text, results):  
 if not results:  
 return set()  
  
 occurrences = [(pos, pos + len(self.patterns[pattern\_num - 1]) - 1, pattern\_num)  
 for pos, pattern\_num in results]  
 overlap\_patterns = set()  
  
 for i in range(len(occurrences)):  
 start1, end1, pattern1 = occurrences[i]  
 for j in range(len(occurrences)):  
 if i != j:  
 start2, end2, pattern2 = occurrences[j]  
 if start1 <= end2 and start2 <= end1:  
 overlap\_patterns.add(pattern1)  
 overlap\_patterns.add(pattern2)  
 return overlap\_patterns  
  
def process\_search(T, patterns):  
 automaton = AhoCorasick(patterns)  
 results = automaton.search(T)  
 vertex\_count = automaton.get\_vertex\_count()  
 overlapping = automaton.find\_overlapping\_patterns(T, results)  
 return results, vertex\_count, overlapping  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 T = input().strip()  
 n = int(input())  
 patterns = [input().strip() for \_ in range(n)]  
  
 results, vertex\_count, overlapping = process\_search(T, patterns)  
  
 print(f"\nКоличество вершин в автомате: {vertex\_count}")  
 for pos, pattern\_num in results:  
 print(pos, pattern\_num)  
 if overlapping:  
 print("Шаблоны с пересечениями:", ", ".join(str(p) for p in sorted(overlapping)))  
 else:  
 print("Шаблоны с пересечениями отсутствуют")

aho\_mask.py

from collections import deque  
  
DEBUG = False  
  
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\_count = 0  
 self.build\_automaton()  
  
 def build\_automaton(self):  
 root = 0  
 self.fail[root] = root  
 self.output\_link[root] = -1  
 self.state\_count = 1  
  
 if DEBUG:  
 print("Построение бора:")  
 print(f"Создано состояние {root} (корень)")  
  
 for i in range(self.num\_patterns):  
 if DEBUG:  
 print(f"\nДобавление шаблона {i + 1}: '{self.patterns[i]}'")  
 current\_state = root  
 for char in self.patterns[i]:  
 if char not in self.transitions[current\_state]:  
 new\_state = self.state\_count  
 self.transitions[current\_state][char] = new\_state  
 if DEBUG:  
 print(f"Создано состояние {new\_state} с переходом из {current\_state} по '{char}'")  
 self.state\_count += 1  
 else:  
 new\_state = self.transitions[current\_state][char]  
 if DEBUG:  
 print(f"Переход из {current\_state} по '{char}' в существующее состояние {new\_state}")  
 current\_state = new\_state  
 self.output[current\_state].append(i)  
 if DEBUG:  
 print(f"Состояние {current\_state} отмечено как конец шаблона {i + 1}")  
  
 if DEBUG:  
 print("\nПостроение суффиксных и конечных ссылок:")  
 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  
 if DEBUG:  
 print(f"Состояние {state}: fail = {root}, output\_link = {self.output\_link[state]}")  
 queue.append(state)  
  
 while queue:  
 current\_state = queue.popleft()  
 for char in self.transitions[current\_state]:  
 next\_state = self.transitions[current\_state][char]  
 queue.append(next\_state)  
  
 if DEBUG:  
 print(f"\nОбрабатываем переход из {current\_state} по '{char}' в {next\_state}")  
 fail\_state = self.fail[current\_state]  
 while fail\_state != root and char not in self.transitions[fail\_state]:  
 if DEBUG:  
 print(f"Состояние {fail\_state} не имеет перехода по '{char}', переходим к fail = {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"Установлена суффиксная ссылка: fail[{next\_state}] = {self.fail[next\_state]}")  
  
 fail = self.fail[next\_state]  
 self.output\_link[next\_state] = fail if self.output[fail] else self.output\_link[fail]  
 if DEBUG:  
 print(f"Установлена конечная ссылка: output\_link[{next\_state}] = {self.output\_link[next\_state]}")  
  
 if DEBUG:  
 print("\nПостроенный автомат:")  
 for state in range(self.state\_count):  
 trans = "{" + ", ".join(f"'{k}': {v}" for k, v in self.transitions[state].items()) + "}"  
 if DEBUG:  
 print(f"Состояние {state}: transitions = {trans}, fail = {self.fail[state]}, "  
 f"output = {self.output[state]}, output\_link = {self.output\_link[state]}")  
  
 def search(self, text):  
 if DEBUG:  
 print("\nПроцесс поиска в тексте:", text)  
 current\_state = 0  
 results = []  
 for i in range(len(text)):  
 char = text[i]  
 if DEBUG:  
 print(f"\nПозиция {i + 1}: символ '{char}', текущее состояние = {current\_state}")  
 while current\_state != 0 and char not in self.transitions[current\_state]:  
 if DEBUG:  
 print(f"Нет перехода по '{char}' из {current\_state}, переходим к fail = {self.fail[current\_state]}")  
 current\_state = self.fail[current\_state]  
  
 if char in self.transitions[current\_state]:  
 next\_state = self.transitions[current\_state][char]  
 if DEBUG:  
 print(f"Переход по '{char}' из {current\_state} в {next\_state}")  
 current\_state = next\_state  
 else:  
 if DEBUG:  
 print(f"Нет перехода по '{char}' из {current\_state}, переходим в корень (0)")  
 current\_state = 0  
  
 temp\_state = current\_state  
 visited = set()  
 while temp\_state != -1 and temp\_state not in visited:  
 visited.add(temp\_state)  
 if self.output[temp\_state]:  
 for pattern\_index in self.output[temp\_state]:  
 pos = i - len(self.patterns[pattern\_index]) + 2  
  
 if DEBUG:  
 print(f"Найдено вхождение шаблона {pattern\_index + 1} на позиции {pos}")  
 results.append((pos, pattern\_index + 1))  
 temp\_state = self.output\_link[temp\_state]  
 return sorted(results)  
  
 def get\_vertex\_count(self):  
 return self.state\_count  
  
 def find\_overlapping\_patterns(self, text, results):  
 if not results:  
 return set()  
  
 occurrences = [(pos, pos + len(self.patterns[pattern\_num - 1]) - 1, pattern\_num)  
 for pos, pattern\_num in results]  
 overlap\_patterns = set()  
  
 for i in range(len(occurrences)):  
 start1, end1, pattern1 = occurrences[i]  
 for j in range(len(occurrences)):  
 if i != j:  
 start2, end2, pattern2 = occurrences[j]  
 if start1 <= end2 and start2 <= end1:  
 overlap\_patterns.add(pattern1)  
 overlap\_patterns.add(pattern2)  
 return overlap\_patterns  
  
def find\_pattern\_with\_wildcards(T, P, wildcard):  
 substrings = [s for s in P.split(wildcard) if s]  
 if not substrings:  
 return [], 0, set()  
  
 k = len(substrings)  
 start\_positions = []  
 pos = 0  
 for i, sub in enumerate(P.split(wildcard)):  
 if sub:  
 start\_positions.append(pos)  
 pos += len(sub) + (1 if i < len(P.split(wildcard)) - 1 else 0)  
  
 ac = AhoCorasick(substrings)  
 matches = ac.search(T)  
  
 n = len(T)  
 C = [0] \* n  
  
 for pos, pattern\_idx in matches:  
 start\_pos\_in\_P = start\_positions[pattern\_idx - 1]  
 text\_start = pos - start\_pos\_in\_P - 1  
 if text\_start >= 0:  
 C[text\_start] += 1  
  
 result = []  
 for i in range(n):  
 if C[i] == k and i + len(P) - 1 < n:  
 valid = True  
 for j in range(len(P)):  
 text\_pos = i + j  
 if P[j] != wildcard and T[text\_pos] != P[j]:  
 valid = False  
 break  
 if valid:  
 result.append(i + 1)  
  
 vertex\_count = ac.get\_vertex\_count()  
 overlapping = ac.find\_overlapping\_patterns(T, matches)  
 return sorted(result), vertex\_count, overlapping  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 T = input().strip()  
 P = input().strip()  
 wildcard = input().strip()  
  
 occurrences, vertex\_count, overlapping = find\_pattern\_with\_wildcards(T, P, wildcard)  
  
 print(f"Количество вершин в автомате: {vertex\_count}")  
 if occurrences:  
 for pos in occurrences:  
 print(pos)  
 else:  
 print(-1)  
 if overlapping:  
 print("Шаблоны с пересечениями:", ", ".join(str(p) for p in sorted(overlapping)))  
 else:  
 print("Шаблоны с пересечениями отсутствуют")