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: Алгоритм Ахо-Корасик

Студент гр. 1384 Усачева Д.В.

Преподаватель Шевелева А. М.

Санкт-Петербург

2023

Цель работы.

Изучить алгоритм Ахо-Корасик.

Задание 1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

Первая строка содержит текст (T, $1 \leq |T| \leq 100000$).

Вторая - число n ($1 \leq n \leq 100000$), каждая следующая из n строк содержит шаблон из набора $P = \{p_1, \ldots, p_n\}$, $1 \leq |p| \leq 75$.

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.

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:

Первая строка содержит текст (T, $1 \leq |T| \leq 100000$).

Вторая строка содержит 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.

Выполнение работы.

Для решения поставленных задачи была реализована программа на языке программирования python.

Был определен класс Vertex для хранения информации о вершине, входящей в бор. Поля класса:

- parent – вершина-родитель текущей вершины;
- symbol – символ, по которому был совершен переход из вершины родителя в текущую;
- transition – словарь, который содержит всевозможные пути из текущей вершины (ключ-символ, по которому совершается переход, значение-вершина, в которую переход был совершен);
- flag – маркер, того, что вершина является конечной для какого-либо образца;
- suffix_link – суффиксная ссылка для данной вершины;
- pattern_number – массив с номерами паттерна (записывается для вершин с фалагом True).

Также был определен класс Trie для обработки и хранения бора. Поля класса:

- trie – бор, список записи добавленных вершин;
- patterns – образцы для поиска, на которые разбивается string_pattern по символу joker;
- length_joker – массив, содержащий количество символов-джокеров в строке;
- string_pattern – строка-образец с джокерами;
- joker – символ-джокер для введенной строки;
- text – текст, в котором ведется множественный поиск образцов;
- result – массив, в котором находится результат выполнения алгоритма Ахо-Корасика.

Поля trie, text, patterns присутствуют в классе для выполнения как первой, так и второй задачи. Остальные поля определены для решения второй задачи.

Методы класса Trie:

- read_data – метод для ввода данных о тексте и образце с джокерами и для разбиения string_pattern по символу joker;
- make_trie – метод для построения бора. Вершина-корень бора определена заранее, а сам бор строится следующим образом: образцы посимвольно добавляются в бор, создавая для каждого неопределенного ранее символа элемент класса Vertex, для вершины сразу определяется родитель и символ, по которому будет совершен переход от родителя. Также сразу определяется и поле transition для вершины-родителя (это позволяет избежать добавления поля ребенка и метода автоматного перехода). Если данная вершина является концом какого-либо образца, то заполняется поле flag и pattern_number;
- find_suffix_link – метод для поиска суффиксной ссылки, если она не определена. В нем реализуется проверка наличия перехода по необходимому символу от суффиксной ссылки родителя, если его не удалось найти, то мы движемся далее по суффиксным ссылкам, пока не достигнем корня;

• get_suffix_link – метод для получения суффиксной ссылки. Если суффиксная ссылка определена или вершина является корнем или ребёнком корня, то метод возвращает эту ссылку. Иначе запускается вышеописанная функция для поиска ссылки;

• algorithm_aho_corasick – метод, в котором реализуется алгоритм Ахо-Корасик. Сначала происходит проверка на наличие пути из вершины по текущему символу. Если переход возможен, то он совершается и происходит проверка флага (является ли текущий символ терминальным состоянием), если же нет возможности совершить переход, то мы рекурсивно двигаемся по суффиксным ссылкам, пока не появится возможность совершить переход или ссылка будет указывать на корень бора. Далее если переход был совершен, происходит проверка на наличие терминального состояния суффиксной ссылки данной вершины и следующих пока не достигнем корня. Если мы попадаем в терминальное состояние, то записываем данные о положение символа в тексте и номер образца в результирующий массив;

• find_patterns – метод для поиска образцов в тексте, с помощью алгоритма Ахо-Корасик.

Вышеперечисленные методы необходимы для решения как первой, так и второй задачи. Для решения первой задачи также реализован метод print_result для корректного вывода результата. Для решения второй задачи образец с джокерами был разбит по символу джокера на подстроки, по которым строился бор. Для обработки полученных данных и были написаны методы:

• find_string_pattern_with_joker – метод для множественного поиска строки-образца с джокерами, использующий данные, полученные в алгоритме Ахо-Корасик. Для оптимизации данного метода результирующий массив (содержит пары – номер образца, индекс вхождения данного образца в текст), полученный с помощью алгоритма Ахо-Корасик был разделен на подмассивы для каждого образца. Проверка образца велась от значений индексов текста подмассива самой короткой длинны, от которых отнималась константа

length_up_to_the_current_pattern – длина строки с джокером до образца с номером выбранного подмассива. Проверка производилась при помощи метода check_the_occurrence возвращающего True или False, если проверка прошла успешно, то индекс добавляется в финальный результирующий массив, которой позже выводится в этой же функции;

- check_the_occurrence – метод для проверки целостности и корректности вхождения строки-образца с джокерами. Данный метод работает рекурсивно проверяя правильность расстояния между соответствующими образцами.

Для описания сложности алгоритма Ахо-Корасик введем обозначения: k – количество вхождений образцов в текст, m – общая длина образцов, l – длина текста. Сложность построения бора составляет O(m), тк добавление ведется посимвольно.

Совпадение с образцом будет обнаружено если флаг вершины равен True или если мы, однако двигаясь по суффиксным ссылкам, мы можем достигнуть одной или нескольких вершин с флагом True. Таким образом, так как мы храним в каждой вершине номер образца (или же список номеров, как в реализации задания 2), оканчивающегося в ней, тогда мы можем для текущего состояния за O(n) найти номера всех образцов, для которых достигнуто совпадение, просто пройдя по суффиксным ссылкам от текущей вершины до корня, где n – отдаленность текущей вершины от корня. Это недостаточно эффективное решение, поскольку в сумме асимптотика получится O(ln) (если все образцы содержатся в тексте максимально возможное количество раз).

Однако если реализовать использование сжатых суффиксных ссылок (не реализовано в данной работе), то для текущей вершины мы сможем за O(1) искать следующее совпадение, в таком случае сложность алгоритма как раз будет равна O(l + k).

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в таблице 1.

Таблица 1 — Результаты тестирования

Входные данные	Выходные данные	Комментарии
NTAG 3 TAGT TAG T	2 2 2 3	Работа программы task_1.py
ACTANCA A$$A$ $	1	Работа программы task_2. py

Выводы.

В ходе выполнения лабораторной работы был изучен алгоритм АхоКорасик, из терминологии изучены такие понятия как суффиксные ссылки, бор и автомат. По своему принципу работы данная программа является реализацией алгоритма Ахо-Корасик. Также разработана программа, выполняющая считывание с клавиатуры исходных данных в виде текста, образца и символа-джокера. Затем программа строит бор по подстрокам, которые были образованы в результате деления образца по символу-джокера, формирует суффиксные ссылки и сжатые суффиксные ссылки для всех узлов бора. Далее программа, работая как автомат, находит позиции вхождений подстрок в исходный текст и фиксирует данные подстроки и позиции. Затем полученные данные обрабатываются и ответ на исходную задачу выводится на экран. По своему принципу работы данная программа является приложением алгоритма Ахо-Корасик.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл: task1.py

```
class Vertex:
    '''
    Класс для хранения информации о вершине, входящей в бор
    parent - вершина-родитель текущей вершины
    symbol - символ, по которому был совершен переход из вершины-родителя в текущую
    transition - словарь, который содержит всевозможные пути из текущей вершины (ключ - символ, по которому совершается
    переход, значение - вершина, в которую переход был совершен)
    flag - маркер, того, что вершина является конечной для какого-либо образца
    suffix_link - суффиксная ссылка для данной вершины
    pattern_number - номер паттерна (записывается для вершин с фалагом True)
    '''
    def __init__(self, parent, symbol):
        self.parent = parent
        self.symbol = symbol
        self.transition = {}
        self.flag = False
        self.suffix_link = None
        self.pattern_number = None


class Trie:
    '''
    Класс для обработки и хранения бора
    trie - бор, список записи добавленных вершин
    patterns - образцы, которые необхоимо найти в тексте
    text - текст, в котором ведется множественный поиск образцов
    '''
    def __init__(self):
        self.trie = [Vertex(None, None)]
        self.patterns = []
        self.text = ''

    '''
    Метод для ввода данных о тексте и образцах
    '''
    def read_data(self):
        self.text = input()
        for i in range(int(input())):
            self.patterns.append(input())

    '''
    Метод для построения бора
    '''
    def make_trie(self):
        self.read_data()
        for pattern in self.patterns:
            vertex = self.trie[0]
            for symbol in pattern:
                if vertex.transition.get(symbol) is not None:
                    vertex = vertex.transition.get(symbol)
                else:
                    last_vertex = Vertex(vertex, symbol)
                    self.trie.append(last_vertex)
```

```
                        vertex.transition.setdefault(symbol, last_vertex)
                        vertex = last_vertex
                vertex.flag = True
                vertex.pattern_number = self.patterns.index(pattern)

        '''
        Метод для поиска суффиксной ссылки, если она не определена
        '''
        def find_suffix_link(self, parent, symbol):
            parent_link = self.get_suffix_link(parent)
            if parent_link == self.trie[0] and
parent_link.transition.get(symbol) is None:
                return self.trie[0]
            else:
                if parent_link.transition.get(symbol) is not None:
                    return parent_link.transition[symbol]
                return self.find_suffix_link(parent_link, symbol)

        '''
        Метод для получения суффиксной ссылки
        '''
        def get_suffix_link(self, vertex):
            if vertex.suffix_link is None:
                if vertex.parent == self.trie[0] or vertex == self.trie[0]:
                    vertex.suffix_link = self.trie[0]
                else:
                    vertex.suffix_link = self.find_suffix_link(vertex.parent,
vertex.symbol)
            return vertex.suffix_link

        '''
        Метод для корректного вывода результата
        '''
        def print_result(self, result):
            result.sort()
            for line in result:
                print(line[0], line[1])

        '''
        Метод, в котором реализуется алгоритм Ахо-Корасик. Сложность
алгоритма описана в отчете.
        '''
        def algorithm_aho_corasick(self, vertex, result, index_in_text):
            if vertex.transition.get(self.text[index_in_text]) is not None:
                vertex = vertex.transition.get(self.text[index_in_text])
                if vertex.flag:
                    result.append(
                        (index_in_text -
len(self.patterns[vertex.pattern_number]) + 2, vertex.pattern_number + 1))
            elif vertex != self.trie[0]:
                vertex = self.get_suffix_link(vertex)
                return self.algorithm_aho_corasick(vertex, result,
index_in_text)
            suffix_link = self.get_suffix_link(vertex)
            while suffix_link != self.trie[0]:
                if suffix_link.flag:
                    result.append((index_in_text -
len(self.patterns[suffix_link.pattern_number]) + 2,
                                   suffix_link.pattern_number + 1))
                suffix_link = self.get_suffix_link(suffix_link)
            return vertex

        '''
```

```
        Метод для поиска образцов в тексте, с помощью алгоритма Ахо-Корасик
        '''
        def find_patterns(self):
            self.make_trie()
            result = []
            vertex = self.trie[0]
            for i in range(len(self.text)):
                vertex = self.algorithm_aho_corasick(vertex, result, i)
            self.print_result(result)


trie = Trie()
trie.find_patterns()
```

Файл: task2.py

```
class Vertex:
    '''
    Класс для хранения информации о вершине, входящей в бор
    parent - вершина-родитель текущей вершины
    symbol - символ, по которому был совершен переход из вершины-родителя в текущую
    transition - словарь, который содержит всевозможные пути из текущей вершины (ключ - символ, по которому совершается
    переход, значение - вершина, в которую переход был совершен)
    flag - маркер, того, что вершина является конечной для какого-либо образца
    suffix_link - суффиксная ссылка для данной вершины
    pattern_number - массив с номерами паттерна (записывается для вершин с фалагом True).
    '''
    def __init__(self, parent, symbol):
        self.parent = parent
        self.symbol = symbol
        self.transition = {}
        self.flag = False
        self.suffix_link = None
        self.pattern_number = []


class Trie:
    '''
    Класс для обработки и хранения бора
    trie - бор, список записи добавленных вершин
    patterns - образцы для поиска, на которые разбивается string_pattern по символу joker
    length_joker - массив, содержащий количество символов-джокеров в строке
    string_pattern - строка-образец с джокерами
    joker - символ-джокер для введенной строки
    text - текст, в котором ведется множественный поиск образцов
    result - массив, в котором находится результат выполнения алгоритма Ахо-Корасика
    '''
    def __init__(self):
        self.trie = [Vertex(None, None)]
        self.patterns = []
        self.length_joker = [0]
        self.string_pattern = ''
        self.joker = ''
        self.text = ''
        self.result = None

    '''
```

```
        Метод для ввода данных о тексте и образце с джокерами и для разбиения
string_pattern по символу joker
        '''
        def read_data(self):
            self.text = input()
            self.string_pattern = input()
            self.joker = input()
            self.patterns = self.string_pattern.split(self.joker)
            for pattern in self.patterns:
                if pattern == '':
                    self.length_joker[-1] += 1
                else:
                    self.length_joker.append(1)
            self.length_joker[-1] -= 1
            self.patterns = list(filter(None, self.patterns))
            self.result = [[] for i in range(len(self.patterns))]

        '''
        Метод для построения бора
        '''
        def make_trie(self):
            self.read_data()
            for i in range(len(self.patterns)):
                pattern = self.patterns[i]
                vertex = self.trie[0]
                for symbol in pattern:
                    if vertex.transition.get(symbol) is not None:
                        vertex = vertex.transition.get(symbol)
                    else:
                        last_vertex = Vertex(vertex, symbol)
                        self.trie.append(last_vertex)
                        vertex.transition.setdefault(symbol, last_vertex)
                        vertex = last_vertex
                vertex.flag = True
                vertex.pattern_number.append(i)

        '''
        Метод для поиска суффиксной ссылки, если она не определена
        '''
        def find_suffix_link(self, parent, symbol):
            parent_link = self.get_suffix_link(parent)
            if        parent_link        ==        self.trie[0]        and
parent_link.transition.get(symbol) is None:
                return self.trie[0]
            else:
                if parent_link.transition.get(symbol) is not None:
                    return parent_link.transition[symbol]
                return self.find_suffix_link(parent_link, symbol)

        '''
        Метод для получения суффиксной ссылки
        '''
        def get_suffix_link(self, vertex):
            if vertex.suffix_link is None:
                if vertex.parent == self.trie[0] or vertex == self.trie[0]:
                    vertex.suffix_link = self.trie[0]
                else:
                    vertex.suffix_link = self.find_suffix_link(vertex.parent,
vertex.symbol)
            return vertex.suffix_link

        '''
```

```
        Метод, в котором реализуется алгоритм Ахо-Корасик. Сложность
алгоритма описана в отчете.
        '''
        def algorithm_aho_corasick(self, vertex, index_in_text):
            if vertex.transition.get(self.text[index_in_text]) is not None:
                vertex = vertex.transition.get(self.text[index_in_text])
                if vertex.flag:
                    for pattern_number in vertex.pattern_number:
                        self.result[pattern_number].append(
                            (pattern_number,         index_in_text         -
len(self.patterns[pattern_number]) + 1))
            elif vertex != self.trie[0]:
                vertex = self.get_suffix_link(vertex)
                return self.algorithm_aho_corasick(vertex, index_in_text)
            suffix_link = self.get_suffix_link(vertex)
            while suffix_link != self.trie[0]:
                if suffix_link.flag:
                    for pattern_number in suffix_link.pattern_number:
                        self.result[pattern_number].append(
                            (pattern_number,         index_in_text         -
len(self.patterns[pattern_number]) + 1))
                suffix_link = self.get_suffix_link(suffix_link)
            return vertex

        '''
        Метод для поиска образцов в тексте, с помощью алгоритма Ахо-Корасик
        '''
        def find_patterns(self):
            self.make_trie()
            vertex = self.trie[0]
            for i in range(len(self.text)):
                vertex = self.algorithm_aho_corasick(vertex, i)
            self.find_string_pattern_with_joker()

        '''
        Метод для множественного поиска строки-образца с джокерами,
использующий данные, полученные в алгоритме Ахо-Корасик
        '''
        def find_string_pattern_with_joker(self):
            final_result = []
            length_array = []
            for i in self.result:
                length_array.append(len(i))
            index_of_minimum_array = length_array.index(min(length_array))
            length_up_to_the_current_pattern = 0
            for i in range(index_of_minimum_array):
                length_up_to_the_current_pattern  +=  self.length_joker[i]  +
len(self.patterns[i])
            length_up_to_the_current_pattern                                 +=
self.length_joker[index_of_minimum_array]
            for current_index in self.result[index_of_minimum_array]:
                start_index          =          current_index[1]          -
length_up_to_the_current_pattern
                if start_index >= 0:
                    if self.check_the_occurrence(start_index, 0):
                        final_result.append(start_index + 1)
            for i in final_result:
                print(i)

        '''
        Метод для проверки целостности и корректности вхождения строки-
образца с джокерами
        '''
```

```
          def check_the_occurrence(self, index, current_number_pattern):
              if current_number_pattern != len(self.patterns):
                  if          (current_number_pattern,          index          +
self.length_joker[current_number_pattern])                                      in
self.result[current_number_pattern]:
                      return         self.check_the_occurrence(index          +
self.length_joker[current_number_pattern] +

len(self.patterns[current_number_pattern]), current_number_pattern + 1)
                  else:
                      return False
              elif   index   +   self.length_joker[current_number_pattern]   <=
len(self.text):
                  return True


      trie = Trie()
      trie.find_patterns()
```