**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МОЭВМ**

отчет

**по лабораторной работе № 5**

**по дисциплине «Построение и Анализ Алгоритмов»**

Тема: «**Ахо-Корасик**»

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Коршков А.А. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

# Цель работы

Написать программы на основе алгоритма Ахо-Корасик для нахождения вхождения всех образцов в строке, а также найти индексы вхождения образцов с джокерами.

# Задания

**№1**

Первая строка содержит текст (T, 1 ≤ ∣T∣ ≤ 100000).

Вторая - число n (1 ≤ n ≤ 3000), каждая следующая из n строк содержит шаблон из набора P={p1,…,pn}1 ≤ ∣pi∣ ≤ 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

**№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? 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

**Задание варианта:**

**7.** Вывод графического представления автомата.

**Примечания для варианта:**

1) В автомате должны быть и использоваться не только суффиксные ссылки, но и конечные ссылки

2) Для обоих заданий на программирование должны быть версии кода с выводом промежуточных данных. В них, в частности, должны выводиться построение бора и автомата, построенный автомат (в виде, например, описания каждой вершины автомата), процесс его использования.

## Основные теоретические положения

**Описание алгоритмов:**

Алгоритм Ахо-Корасик называют ещё «расширенной версией КМП». КМП очень быстро может найти вхождение строки, но если строк слишком много, то . Алгоритм Ахо-Корасик позволяет.

Алгоритм создает префиксное дерево из букв искомых подстрок. Вершины, в которых искомая подстрока заканчивается называет терминальной и выделяется специальным цветом в графическом представлении. Суффиксная ссылка вершины u – это вершина v, такая что строка v является максимальным суффиксом строки u. Для корня и вершин, исходящих из корня, суффиксной ссылкой является корень. Для остальных вершин осуществляется переход по суффиксной ссылке родителя и, если оттуда есть ребро с заданным символом, суффиксная ссылка назначается в вершину, куда это ребро ведет. Cуффиксные сслыки находятся не автоматически для каждой вершины, а вычисляются во время работы программы при обращении к специальному методу.

**Оценка сложности по памяти и операциям**

Сложность по времени в наихудшем случае O((N+1) (M+1)), т.к. необходимо пройтись по всей таблице.

Сложность по памяти O((N+1) (M+1)), т.к. необходимо хранить матрицу размером (n+1) \* (m+1).

Примечание: для нахождения последовательностей операций необходимо хранить вторую таблицу с наилучшими выборами операции на каждом шаге. Поэтому в этом случае сложность по памяти .

## Выполнение работы

**Описание работы**

Для решения заданий были написаны два класса Vertex и Trie, которые представляют вершину автомата и сам бор.

В классе Vertex описаны следующие методы:

\_\_init\_\_(self, id\_: int, alpha: int, parent: "Vertex" or None = None, pchar: str or None = None) -> None – констурктор класса вершины Vertex. В качестве аргументов принимает номер вершины, размер алфавита, родительскую вершину (если есть), символ родительской вершины (если есть).

is\_terminal(self) -> bool – возвращает True, если вершина является терминальной

@is\_terminal.setter

is\_terminal(self, value: bool) -> None – позволяет установить флаг для терминальной вершины

id(self) -> int – возвращает присвоенный идентификатор вершины

sufflink(self) -> "Vertex" or None – возвращает суффиксную ссылку на вершину, если суффиксная ссылка была вычислена для данной вершины.

@sufflink.setter

sufflink(self, value) -> None - позволяет установить значение для суффиксной ссылки

parent(self) -> "Vertex" or None – возвращает родительскую вершину, если это не корень бора.

pchar(self) -> str or None – возвращает символ родительской вершины, если это не корень бора.

\_\_str\_\_(self) -> str – возвращает информацию в строковом виде для вершины

В классе Trie описаны следующие методы:

\_\_init\_\_(self, alpha: int = 5) -> None – конструктор для класса автомата Ахо-Корасик. На вход принимает размер алфавита (по умолчанию 5, для заданного алфавита {A: 0, C: 1, G: 2, T: 3, N: 4})

size(self) -> int – возвращает кол-во вершин в дереве

last(self) -> Vertex – ввозвращает последнюю вершину в дереве

alpha(self) -> int – возвращает размер алфавита

vertices(self) -> list[Vertex] – возвращает список вершин

root(self) -> возвращает корень дерева

add(self, s: str, pattern\_num: int) -> None – добавляет образец в дерево

search(self, s: str) -> list[tuple[int, int]] - проверяет, есть ли строка в дереве и возвращает

get\_link(self, v: Vertex) -> Vertex – находит и возвращает суффиксную ссылку для вершины.

go(self, v: Vertex, char: str) -> Vertex - Возвращает вершину, в которую ведет переход по символу char из вершины v.

precompute\_sufflinks(self) -> None – предварительно вычисляет все суффиксные ссылки (нужно для визуализации автомата)

visualize(self, file\_name: str = "aho\_corasick") -> None - Создает графическое представление автомата Ахо-Корасик и сохраняет его в png файл. Также создаётся легенда для графа. На вход принимает имя файла, в который нужно сохранить визуализацию.

Также для класса Trie есть вспомогательные внутренние методы:

\_num(c: str) -> int – возвращает номер буквы в заданном алфавите из 5 букв: {A: 0, C: 1, G: 2, T: 3, N: 4}

\_char(idx: int) -> int – возвращает символ буквы, соответствующий номеру в алфавите.

В файла main.py есть несколько функций для решения заданий:

main() -> None – главная функция, которая запускает функции для решения заданий

aho\_corasick\_search() -> None – функция, которая ищет позиции вхождения всех заданных образцов в тексте.

search\_with\_wildcard() -> None – функция, которая ищет индексы вхождения образца с джокером.

# Тестирование

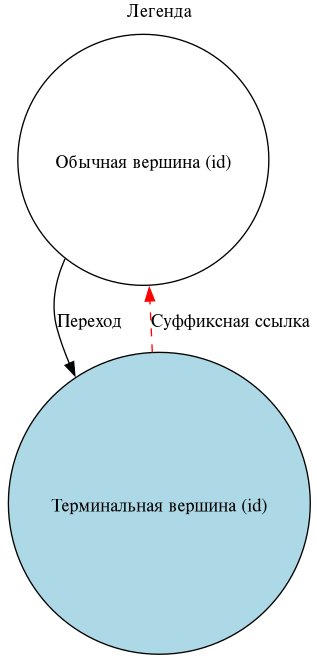


Рисунок – Легенда графа

Таблица 1 – Тестирование алгоритмов

|  |  |  |  |
| --- | --- | --- | --- |
| № | Входные данные | Выходные данные | Комментарии |
| 1 | NTAG  3  TAGT  TAG  T | 2 2  2 3 | Алгоритм Ахо-Корасик. Верно |
| 2 | ACGT  3  ACGT  CG  GT | 1 1  2 2  3 3 | Алгоритм Ахо-Корасик. Верно |
| 3 |  |  | Алгоритм Ахо-Корасик. Верно |
| 4 | ACTANCA  A$$A$  $ | 1 | Ахо-Корасик с джокером. |
| 5 |  | 7 | Ахо-Корасик с джокером. |
| 6 |  | 5 | Ахо-Корасик с джокером. |

# Выводы

Был реализован алгоритм Вагнера-Фишера для расстояния Левенштейна и редакционного предписания между двумя строками, определяя минимальное количество операций (вставки, удаления, замены, *вставка двух одинаковых символов подряд*) для преобразования одной строки в другую.

ПРИЛОЖЕНИЕ А  
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

"""

Главный файл программы.

Вар. 3а. Добавляется 4-я операция со своей стоимостью: последовательная вставка

двух одинаковых символов.

"""

from modules.vagner\_fisher import calculate\_edit\_distance, compute\_edit\_sequence

def main() -> None:

"""

Главная функция

:return:

"""

print("Задание #1: Алгоритм Вагнера-Фишера")

rep\_cost, ins\_cost, del\_cost, ins2\_cost = map(int, input().split())

s1: str = input()

s2: str = input()

print("Результат:", calculate\_edit\_distance(s1, s2, rep\_cost, ins\_cost, del\_cost, ins2\_cost))

print("Задание #2: Алгоритм Вагнера-Фишера. Порядок операции")

rep\_cost, ins\_cost, del\_cost, ins2\_cost = map(int, input().split())

s1: str = input()

s2: str = input()

print(compute\_edit\_sequence(s1, s2, rep\_cost, ins\_cost, del\_cost, ins2\_cost), s1, s2, sep="\n")

print("Задание #3: Расстояние Левенштейна")

s1: str = input()

s2: str = input()

print("Результат:", calculate\_edit\_distance(s1, s2))

if \_\_name\_\_ == "\_\_main\_\_":

main()

Название файла: vagner\_fisher.py

"""

Модуль для вычисления алгоритма Вагнера-Фишера.

"""

def \_wagner\_fisher\_step(i: int, j: int, s1: str, s2: str, matrix: list[list[int]],

rep\_cost: int, ins\_cost: int, del\_cost: int, ins2\_cost: int) -> int:

"""

Вычисляет шаг алгоритма Вагнера-Фишера для двух строк s1 и s2.

:param i:

:param j:

:param s1:

:param s2:

:param matrix:

:param rep\_cost:

:param ins\_cost:

:param del\_cost:

:param ins2\_cost:

:return:

"""

print(f"--- Вычисление ячейки ({i}, {j}) ---")

if i == 0 and j == 0:

print("Начальная ячейка (0, 0), значение 0.")

return 0

if j == 0:

val: int = i \* del\_cost

print(f"j=0: удаление {i} символов. Значение = {val}.")

return val

if i == 0:

if j == 1:

val: int = matrix[0][0] + ins\_cost

print(f"i=0, j=1: единичная вставка. Значение = {val} (0 + {ins\_cost}).")

return val

val1: int = matrix[0][j - 1] + ins\_cost

val2: int = matrix[0][j - 2] + ins2\_cost

val: int = min(val1, val2)

print(f"i=0, j={j}: варианты {val1} (одиночная вставка) "

f"и {val2} (двойная вставка). "

f"Минимум: {val}.")

return val

# Замена или совпадение

rep: int = matrix[i - 1][j - 1] + (0 if s1[i - 1] == s2[j - 1] else rep\_cost)

ins: int = matrix[i][j - 1] + ins\_cost

dele: int = matrix[i - 1][j] + del\_cost

candidates: list[int] = [rep, ins, dele]

if j >= 2:

ins2\_val = matrix[i][j - 2] + ins2\_cost

print(f"Двойная вставка: {ins2\_val} (база {matrix[i][j - 2]} + {ins2\_cost})")

candidates.append(ins2\_val)

else:

print("Двойная вставка недоступна (j < 2)")

print(f"Кандидаты для ячейки ({i}, {j}): {candidates}. "

f"Минимальное значение: {min(candidates)}.")

return min(candidates)

def calculate\_edit\_distance(s1: str, s2: str,

rep\_cost: int = 1, ins\_cost: int = 1,

del\_cost: int = 1, ins2\_cost: int = 1) -> int:

"""

Вычисляет расстояние редактирования между строками s1 и s2

с учётом операций: замены, вставки, удаления и

последовательной вставки двух одинаковых символов.

:param s1:

:param s2:

:param rep\_cost:

:param ins\_cost:

:param del\_cost:

:param ins2\_cost:

:return:

"""

n, m = len(s1), len(s2)

matrix: list[list[int]] = [[0] \* (m + 1) for \_ in range(n + 1)]

for i in range(n + 1):

for j in range(m + 1):

matrix[i][j] = \_wagner\_fisher\_step(i, j, s1, s2, matrix,

rep\_cost, ins\_cost,

del\_cost, ins2\_cost)

print(f"Текущее значение матрицы[{i}][{j}] = {matrix[i][j]}")

print(f"\nСостояние матрицы после строки i={i}:")

for row in matrix[:i + 1]:

print(' '.join(map(str, row)))

print("-" \* 50 + "\n")

return matrix[n][m]

def compute\_edit\_sequence(s1: str, s2: str,

rep\_cost: int = 1, ins\_cost: int = 1,

del\_cost: int = 1, ins2\_cost: int = 1) -> str:

"""

Вычисляет последовательность операций для преобразования строки s1 в s2

с учётом дополнительных затрат при последовательной вставке двух символов.

Обозначения:

M – совпадение

R – замена

I – вставка одного символа

D – удаление символа

P – последовательная вставка двух одинаковых символов

:param s1:

:param s2:

:param rep\_cost:

:param ins\_cost:

:param del\_cost:

:param ins2\_cost:

:return:

"""

n, m = len(s1), len(s2)

cost: list[list[int]] = [[0] \* (m + 1) for \_ in range(n + 1)]

back: list[list[str]] = [[''] \* (m + 1) for \_ in range(n + 1)]

print("\n" + "=" \* 50)

print("Инициализация первого столбца (операции удаления):")

for i in range(1, n + 1):

cost[i][0]: int = cost[i - 1][0] + del\_cost

back[i][0]: str = 'D'

print(f"\ti={i}, j=0 → УДАЛЕНИЕ (D). cost[{i}][0] = {cost[i][0]} "

f"(предыдущее {cost[i-1][0]} + {del\_cost})")

print("\n" + "=" \* 50)

print("Инициализация первой строки (операции вставки):")

if m >= 1:

cost[0][1]: int = cost[0][0] + ins\_cost

back[0][1]: str = 'I'

print(f"\ti=0, j=1 → ВСТАВКА (I). cost[0][1] = {cost[0][1]} (0 + {ins\_cost})")

for j in range(2, m + 1):

candidate\_single: int = cost[0][j - 1] + ins\_cost

candidate\_double: int = cost[0][j - 2] + ins2\_cost

print(f"\n i=0, j={j}:")

print(f"\tВариант 1: одиночная вставка → {candidate\_single} "

f"(cost[0][{j - 1}]={cost[0][j - 1]} + {ins\_cost})")

print(f"\tВариант 2: двойная вставка → {candidate\_double} "

f"(cost[0][{j - 2}]={cost[0][j - 2]} + {ins2\_cost})")

if candidate\_double < candidate\_single:

cost[0][j]: int = candidate\_double

back[0][j]: str = 'P'

print("\tВыбрана ДВОЙНАЯ ВСТАВКА (P)")

else:

cost[0][j]: int = candidate\_single

back[0][j]: str = 'I'

print("\tВыбрана ОДИНОЧНАЯ ВСТАВКА (I)")

print(f"\tcost[0][{j}] = {cost[0][j]}, back[0][{j}] = '{back[0][j]}'")

print("\n" + "=" \* 50)

print("Заполнение основной матрицы:")

for i in range(1, n + 1):

print(f"\nОбработка строки i={i}:")

for j in range(1, m + 1):

print(f"\n--- Ячейка ({i}, {j}) ---")

print(f"\tСимволы: s1[{i - 1}] = '{s1[i - 1]}', s2[{j - 1}] = '{s2[j - 1]}'")

if s1[i - 1] == s2[j - 1]:

rep\_val: int = cost[i - 1][j - 1]

op\_rep: str = 'M'

print(f"\tСОВПАДЕНИЕ (M): cost = {rep\_val}")

else:

rep\_val: int = cost[i - 1][j - 1] + rep\_cost

op\_rep: str = 'R'

print(f"\tЗАМЕНА (R): cost = {cost[i - 1][j - 1]} + {rep\_cost} = {rep\_val}")

ins\_val: int = cost[i][j - 1] + ins\_cost

op\_ins: str = 'I'

print(f"\tВСТАВКА (I): cost = {cost[i][j - 1]} + {ins\_cost} = {ins\_val}")

del\_val: int = cost[i - 1][j] + del\_cost

op\_del: str = 'D'

print(f"\tУДАЛЕНИЕ (D): cost = {cost[i - 1][j]} + {del\_cost} = {del\_val}")

best: int = rep\_val

best\_op: str = op\_rep

if ins\_val < best:

best: int = ins\_val

best\_op: str = op\_ins

if del\_val < best:

best: int = del\_val

best\_op: str = op\_del

if j >= 2:

double\_ins\_val: int = cost[i][j - 2] + ins2\_cost

print(f"\tДВОЙНАЯ ВСТАВКА (P): cost = {cost[i][j - 2]} + {ins2\_cost} = {double\_ins\_val}")

if double\_ins\_val < best:

best: int = double\_ins\_val

best\_op: str = 'P'

cost[i][j]: int = best

back[i][j]: str = best\_op

print(f"\tВыбранная операция: '{best\_op}' → cost[{i}][{j}] = {best}")

print("\n" + "=" \* 50)

print("Матрица стоимостей операций")

for row in cost:

print('\t'.join(map(str, row)))

print("\n" + "=" \* 50)

print("Матрица оптимальных операций")

for row in back:

print('\t'.join(row))

print("\n" + "=" \* 50)

print("Восстановление последовательности операций:")

i, j = n, m

operations: list = []

while i > 0 or j > 0:

op: str = back[i][j]

operations.append(op)

print(f"\tПозиция ({i}, {j}): операция '{op}'")

if op in ('M', 'R'):

i -= 1

j -= 1

elif op == 'I':

j -= 1

elif op == 'D':

i -= 1

elif op == 'P':

j -= 2

operations.reverse()

return ''.join(operations)