**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. 3341 |  | Ягудин Д.Р. |
| Преподаватель |  | Фирсов М.А. |

# 

# Задание 1.

Разработайте программу, решающую задачу точного поиска набора образцов.

**Вход:**

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

Вторая — число *n* (1≤*n*≤3000), каждая следующая из *n* строк содержит шаблон из набора *P*={*p*1,…,*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

# 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 а*b*??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

# Индивидуализация:

# Вывод графического представления автомата.

# Основные теоретические положения.

Пусть дан набор строк в алфавите размера k суммарной длины m. Алгоритм Ахо-Корасик строит для этого набора строк структуру данных "бор", а затем по этому бору строит автомат, всё за O(m) времени и O(mk) памяти. Полученный автомат уже может использоваться в различных задачах, простейшая из которых — это нахождение всех вхождений каждой строки из данного набора в некоторый текст за линейное время.

*Сложности алгоритма* по операциям O(nA + T + k), где n – общая длина всех слов в словаре, A – размер алфавита, T – длина текста, в котором проводится поиск, k – общая длина всех совпадений

# Выполнение работы.

**1. Визуализация структур данных**

Для визуализации структур данных реализованы две функции:

*visualize\_trie(root, filename)* - визуализирует префиксное дерево (бор) алгоритма Ахо-Корасик с помощью Graphviz. Отображает узлы с их идентификаторами, выходными значениями и fail-ссылками.

*visualize\_wildcard\_automaton(root, filename)*- визуализирует автомат для поиска с джокером, аналогично отображая узлы с их позициями и fail-ссылками.

**2. Классический алгоритм Ахо-Корасик**

Класс TrieNode

Для представления вершины бора реализован класс TrieNode, содержащий следующие поля:

*children*- словарь, хранящий переходы по символам к дочерним узлам;

*fail* - fail-ссылка (аналог суффиксной ссылки);

*output* - множество идентификаторов шаблонов, которые заканчиваются в данной вершине;

*id*- уникальный идентификатор узла.

Методы класса:

*\_\_repr\_\_* - строковое представление узла для отладки.

Функции алгоритма:

*build\_trie(patterns, debug)* - строит бор на основе набора шаблонов;

*build\_failure\_links(root, debug)* - добавляет fail-ссылки к каждой вершине бора;

*aho\_corasick\_search(text, patterns, debug)* - реализует поиск всех вхождений шаблонов в текст с использованием построенного автомата.

**3. Алгоритм поиска с джокером (wildcard)**

Класс WildcardNode

Для представления вершины автомата реализован класс WildcardNode, содержащий следующие поля:

*children* - словарь переходов по символам;

*positions* - список позиций в шаблоне, соответствующих данному узлу;

*fail* - fail-ссылка;

*id* - уникальный идентификатор узла.

Методы класса:

*\_\_repr\_\_* - строковое представление узла для отладки.

Функции алгоритма:

*build\_wildcard\_automaton(pattern, wildcard, debug)* - строит автомат для поиска шаблона с джокерами;

*wildcard\_search(text, pattern, wildcard, debug)* - выполняет поиск шаблона с джокерами в тексте.

# Тестирование.

*Задание1.*

# Входные данные:

NTAG 3 TAGT TAG T

# Выходные данные:

2 2

2 3

# Входные данные:

ACGT 3 CGTA GT

A

# Выходные данные:

1 3

3 2

# Входные данные:

GGTA 3 GTA AG

T

# Выходные данные:

2 1

3 3

*Задание2.*

1. Входные данные:

ACTANCA

A$$A$

$

Выходные данные:

1

1. Входные данные:

CGTTTTNCGAS

CG\*\*

\*

Выходные данные:

1

8

# Входные данные:

TNAAAGCAAGAAG

AA& &

# Выходные данные:

3

4

8

11

# Выводы.

В ходе работы был изучен алгоритм Ахо-Корасик поиска набора образцов в строке, применён в решении поставленной задачи на языке программирования Python. Реализовано тестирование программы.

**ПРИЛОЖЕНИЕ А ИСХОДНЫЙ КОД ПРОГРАММЫ**

Название файла: main.py

import src.includes as lib

while(True):

print("what do you want to use(aho/wild/exit)")

user\_input = input()

if (user\_input == "exit"):

break

if (user\_input != "aho" and user\_input != "wild"):

print("incorrect input")

continue

print("do you wanna see how program work(1/0)")

break\_down\_flag = int(input())

if (user\_input == "aho"):

print("input text")

text = input()

print("input sub strings count")

n = int(input())

print("input sub strings")

subs = []

for i in range(n):

subs.append(input())

result = lib.aho\_corasick\_search(text, subs, break\_down\_flag)

for pos,num in result:

print(pos, num)

elif (user\_input =="wild"):

print("input text")

text = input()

print("input sub string")

sub = input()

print("input wildcard char")

wild\_card = input()

result = lib.wildcard\_search(text, sub, wild\_card,break\_down\_flag)

for pos in result:

print(pos)

Название файла: includes.py

import sys

from collections import deque, defaultdict

from graphviz import Digraph

def visualize\_trie(root, filename='trie'):

    """Визуализация префиксного дерева Ахо-Корасик"""

    dot = Digraph(comment='Trie')

    visited = set()

    def add\_nodes(node):

        if node.id in visited:

            return

        visited.add(node.id)

        label = (

            f"ID: {node.id}\n"

            f"Output: {node.output if node.output else '∅'}\n"

            f"Fail: {node.fail.id if node.fail else '∅'}"

        )

        dot.node(str(node.id), label)

        for char, child in node.children.items():

            add\_nodes(child)

            dot.edge(str(node.id), str(child.id), label=char)

        if node.fail and node.fail != node:

            dot.edge(str(node.id), str(node.fail.id), style='dashed', color='red')

    add\_nodes(root)

    dot.render(filename, view=False, format='png', cleanup=True)

    print(f"Граф сохранен в {filename}.png")

def visualize\_wildcard\_automaton(root, filename='wildcard\_automaton'):

    """Визуализация автомата для поиска с джокером"""

    dot = Digraph(comment='Wildcard Automaton')

    visited = set()

    def add\_nodes(node):

        if node.id in visited:

            return

        visited.add(node.id)

        label = (

            f"ID: {node.id}\n"

            f"Positions: {node.positions if node.positions else '∅'}\n"

            f"Fail: {node.fail.id if node.fail else '∅'}"

        )

        dot.node(str(node.id), label, shape='rectangle')

        for char, child in node.children.items():

            add\_nodes(child)

            dot.edge(str(node.id), str(child.id), label=char)

        if node.fail and node.fail != node:

            dot.edge(str(node.id), str(node.fail.id), style='dashed', color='blue')

    add\_nodes(root)

    dot.render(filename, view=False, format='png',cleanup=True)

    print(f"Граф сохранен в {filename}.png")

# Реализация классического алгоритма Ахо-Корасик для множественного поиска

class TrieNode:

    \_id = 0

    \_\_slots\_\_ = ['children', 'fail', 'output', 'id']

    def \_\_init\_\_(self):

        self.children = dict()

        self.fail = None

        self.output = set()

        self.id = TrieNode.\_id

        TrieNode.\_id += 1

    def \_\_repr\_\_(self):

        return f"[Trie#{self.id} ch={list(self.children.keys())} out={self.output} fail={self.fail.id if self.fail else 'root'}]"

def build\_trie(patterns, debug=False):

    TrieNode.\_id = 0  # Сбрасываем счетчик для повторных запусков

    root = TrieNode()

    for idx, pattern in enumerate(patterns, 1):

        if debug: print(f"\nДобавление шаблона {idx}: '{pattern}'")

        node = root

        for char in pattern:

            if char not in node.children:

                node.children[char] = TrieNode()

                if debug: print(f"  Создан узел {node.children[char]}")

            node = node.children[char]

        node.output.add(idx)

        if debug: print(f"  Конец шаблона -> узел {node}")

    return root

def build\_failure\_links(root, debug=False):

    queue = deque()

    root.fail = root

    for child in root.children.values():

        child.fail = root

        queue.append(child)

    while queue:

        current = queue.popleft()

        if debug: print(f"\nОбработка узла {current}")

        for char, child in current.children.items():

            fail\_node = current.fail

            while fail\_node != root and char not in fail\_node.children:

                fail\_node = fail\_node.fail

            child.fail = fail\_node.children.get(char, root)

            child.output.update(child.fail.output)

            if debug:

                fail\_info = f"{child.fail.id}" if child.fail != root else "root"

                print(f"  Установка fail[{char}]: {child.id} → {fail\_info}")

            queue.append(child)

    return root

def aho\_corasick\_search(text, patterns, debug=False):

    root = build\_trie(patterns, debug)

    root = build\_failure\_links(root, debug)

    matches = []

    current = root

    for i, char in enumerate(text, 1):

        if debug:

            print(f"\nProcessing text char '{char}' at position {i}")

        while current != root and char not in current.children:

            if debug:

                print(f"  Following fail link from {current} to {current.fail}")

            current = current.fail

        if char in current.children:

            current = current.children[char]

            if debug:

                print(f"  Moved to node {current}")

        else:

            if debug:

                print("  No transition, stay at root")

        if current.output:

            if debug:

                print(f"  Output patterns: {current.output}")

            for pattern\_id in current.output:

                start = i - len(patterns[pattern\_id - 1]) + 1

                matches.append((start, pattern\_id))

    print("input visual png name")

    filename = input()

    visualize\_trie(root, filename)

    return sorted(matches)

# Реализация поиска с джокером

class WildcardNode:

    \_id = 0

    \_\_slots\_\_ = ['children', 'positions', 'fail', 'id']

    def \_\_init\_\_(self):

        self.children = dict()

        self.positions = list()

        self.fail = None

        self.id = WildcardNode.\_id

        WildcardNode.\_id += 1

    def \_\_repr\_\_(self):

        return f"[Wild#{self.id} ch={list(self.children.keys())} pos={self.positions} fail={self.fail.id if self.fail else 'root'}]"

def build\_wildcard\_automaton(pattern, wildcard, debug=False):

    WildcardNode.\_id = 0  # Сбрасываем счетчик

    required = [(j, c) for j, c in enumerate(pattern) if c != wildcard]

    if debug:

        print("\nПостроение автомата для шаблона:")

        print(f"Шаблон: {pattern} (джокер: '{wildcard}')")

        print(f"Ключевые позиции: {required}")

    if not required:

        if debug: print("Нет ключевых символов!")

        return None

    root = WildcardNode()

    char\_map = defaultdict(list)

    for j, c in required:

        char\_map[c].append(j)

    for c, positions in char\_map.items():

        if debug: print(f"\nДобавление символа '{c}' на позициях {positions}")

        node = root

        if c not in node.children:

            node.children[c] = WildcardNode()

            if debug: print(f"  Создан узел {node.children[c]}")

        node = node.children[c]

        node.positions.extend(positions)

        if debug: print(f"  Обновлен узел {node}")

    queue = deque()

    root.fail = root

    for child in root.children.values():

        child.fail = root

        queue.append(child)

    while queue:

        current = queue.popleft()

        if debug: print(f"\nОбработка узла {current}")

        for char, child in current.children.items():

            fail\_node = current.fail

            while fail\_node != root and char not in fail\_node.children:

                fail\_node = fail\_node.fail

            child.fail = fail\_node.children.get(char, root)

            if debug:

                fail\_info = f"{child.fail.id}" if child.fail != root else "root"

                print(f"  Установка fail[{char}]: {child.id} → {fail\_info}")

            queue.append(child)

    return root, len(pattern), required

def wildcard\_search(text, pattern, wildcard, debug=False):

    automaton = build\_wildcard\_automaton(pattern, wildcard, debug)

    if not automaton: return []

    root, pat\_len, required = automaton

    if debug:

        print("\nНачало поиска в тексте:")

        print(f"Текст: {text}")

        print(f"Длина шаблона: {pat\_len}")

        print(f"Требуемые совпадения: {required}\n")

    current = root

    checked = set()

    result = set()

    for pos, char in enumerate(text):

        if debug: print(f"\nПозиция {pos}: символ '{char}'")

        # Переход по fail-ссылкам

        steps = 0

        while current != root and char not in current.children:

            if debug:

                print(f"  Шаг {steps}: переход {current.id} → {current.fail.id}")

                steps += 1

            current = current.fail

        # Обработка текущего символа

        if char in current.children:

            current = current.children[char]

            if debug: print(f"  Переход в узел {current}")

        else:

            if debug: print("  Нет перехода, остаемся в root")

            current = root

        # Проверка всех возможных позиций

        temp = current

        depth = 0

        while temp != root:

            if temp.positions:

                if debug: print(f"  Глубина {depth}: узел {temp} содержит позиции {temp.positions}")

                for j in temp.positions:

                    start = pos - j

                    if debug: print(f"    Проверка позиции {start} (j={j})")

                    if start < 0 or start + pat\_len > len(text):

                        if debug: print("      Выход за границы текста")

                        continue

                    if start in checked:

                        if debug: print("      Уже проверено")

                        continue

                    checked.add(start)

                    valid = all(text[start+jj] == cc for jj, cc in required)

                    if debug:

                        substr = text[start:start+pat\_len]

                        marks = "".join("ok" if text[start+jj] == cc else "false"

                                      for jj, cc in required)

                        print(f"      Подстрока: {substr} Проверки: {marks}")

                    if valid:

                        result.add(start + 1)

                        if debug: print("       Совпадение!")

            temp = temp.fail

            depth += 1

    print("input visual png name")

    filename = input()

    visualize\_wildcard\_automaton(root, filename)

    return sorted(result)