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. 3343 _____ Лихацкий В. Р,

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы.

Изучить принцип работы алгоритма Кнута-Морриса_Пратта. Написать функцию, вычисляющую для каждого элемента строки максимальное значение длины префикса и с помощью данной функции решить поставленные задачи. А именно написать программу, осуществляющую поиск вхождений подстроки в строку, а также программу, определяющую, являются ли строки циклическим сдвигом друг друга, найти индекс начала вхождения второй строки в первую.

Задание №1.

Разработайте программу, решающую задачу точного поиска набора образцов.

Вход:

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

Вторая - число n*n* (1≤n≤30001≤*n*≤3000), каждая следующая из n*n* строк содержит шаблон из набора P={p1,…,pn}1≤|pi|≤75*P*={*p*1,…,*pn*}1≤|*pi*|≤75

Все строки содержат символы из алфавита {A,C,G,T,N}{*A*,*C*,*G*,*T*,*N*}

Выход:

Все вхождения образцов из P*P* в T*T*.

Каждое вхождение образца в текст представить в виде двух чисел - i *i* p*p*

Где i*i* - позиция в тексте (нумерация начинается с 1), с которой начинается вхождение образца с номером p*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*P* необходимо найти все вхождения PP в текст TT.

Например, образец ab??c?a*b*??c? с джокером ?? встречается дважды в тексте *xabvccbababcax*.

Символ джокер не входит в алфавит, символы которого используются в T*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}{*A*,*C*,*G*,*T*,*N*}

Вход:

Текст (T,1≤|T|≤100000*T*,1≤|*T*|≤100000)

Шаблон (P,1≤|P|≤40*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.

SampleInput:

ACTANCA

A$$A$

$

Sample Output:

1

Вариант 1. На месте джокера может быть любой символ, за исключением заданного.

Описание алгоритмов.

Описание алгоритма Ахо-Корасик.

Алгоритм создает префиксное дерево из букв искомых подстрок. Затем в полученном дереве ищутся суффиксные ссылки. 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. Далее создаются терминальные ссылки – такие суффиксные ссылки, которые ведут в вершину, которая является терминальной.

Текст, в котором нужно найти подстроки побуквенно передается в автомат. Начиная из корня, автомат переходит по ребру, соответствующему переданному символу. Если нужного ребра нет, переходит по ссылке. Если встреченная вершина является терминальной, значит была встречена

подстрока. Если найдено совпадение нужно пройти по терминальным ссылкам, если они не None, чтобы вывести все шаблоны заканчивающиеся на этом месте.Номер подстроки (подстрок) хранится в поле *terminate*вершины. В ответ сохранятся индекс, на котором началась эта подстрока в тексте и сам номер подстроки.

Сложность по времени:

Т.к при построении префиксного дерева запускается цикл по длине каждой подстроки (суммарная длина подстрок - n), и из каждой вершины может исходить максимум k ребер (где k – размер алфавита), то построение префиксного дерева происходит за O(n*k)

Алгоритм в цикле проходит по тексту длины s: O(s)

Итого: O(n*k + s)

Сложность по памяти:

Алгоритм создает префиксное дерево с n вершинами, каждая вершина хранит массив вершин, инцидентных ей, размером k (k – размер алфавита).

Итого: O(n*k) Описание алгоритма для нахождения шаблонов с маской.

Описание модифицированного алгоритма.

Алгоритм тот же, но в качестве подстрок берутся кусочки шаблона, разделенные джокером, запоминаются позиции полученных подстрок в исходном шаблоне. Создается массив C длины s, где s – длина текста, где ищется шаблон. При нахождении подстроки, в массиве C увеличивается на единицу число по индексу, соответствующему возможному началу шаблона. Индекс высчитывается по формуле: текущий индекс - (длина найденной подстроки - 1) - (позиция подстроки в шаблоне -1). Затем проходим по полученному массиву, каждый i для которого C[i] = количеству подстрок, является вероятным началом шаблона. В соответствии с индивидуализацией, для каждого найденного шаблона проверяются буквы, стоящие на месте джокера. Если не было встречено запрещенного символа, найденный шаблон добавляется в ответ.

Сложность по времени для модифицированного алгоритма:

Затраты по времени такие же как в обычном алгоритме, но дополнительно проход по массиву C длины s: Итого: O(n*k + s + s) = O(n*k + s + t)

Сложность по памяти для модифицированного алгоритма:

Затраты по памяти такие же как в обычном алгоритме, но дополнительно создается массив C длины s. Затраты по памяти O(n*k + s)

Описание функций.

1. **Структура Node**: Представляет узел в префиксном дереве (trie), содержащий ссылки на, детей, суффиксные и терминальные ссылки.
2. **Класс SearchTree**: Хранит бор и конечный автомат , содержит методы для поиска подстрок в тексте
    1. **addPattern(pattern: Pattern):** Добавляет паттерн в дерево поиска, конец паттерна помечает.
    2. **setLinks()**: Обходит дерево в ширину и создает суффиксные и терминальные ссылки для всех узлов, начиная с корня.
    3. **search(text string) SearchEntry[]** Реализует алгоритм Ахо-Корасик для поиска всех вхождений шаблонов в тексте, используя суффиксные и терминальные ссылки
3. **Тип Pattern –** содержит строковое представление шаблона, его индекс
4. **Тип SearchEntry –** содержит найденное совпадение в тексте: pattern и индекс совпадения at.

Тестирование.

Входные данные	Ответ	Комментарий
NTAG 3 TAGT TAG T	2 2 2 3	Верно
ACCGTACA 2 AC GT	1 1 4 2 6 1	Верно
ACGT 3 ACGT CG GT	1 1 2 2 3 3	Верно

Таблица 1 – Тестирование алгоритма Ахо-Корасик

Входные данные	Ответ	Комментарий
ACTANCA A$$A$ $ G	1	Верно
ACACAA ACXA X Y	3	Верно
ACGANGAAAT A$G $ C	4	Верно

Таблица 1 – Тестирование алгоритма поиска с джокером

Результат работы программы с отладочным выводом для первого задания (см. рис 1, 2, 3).

Исследование.

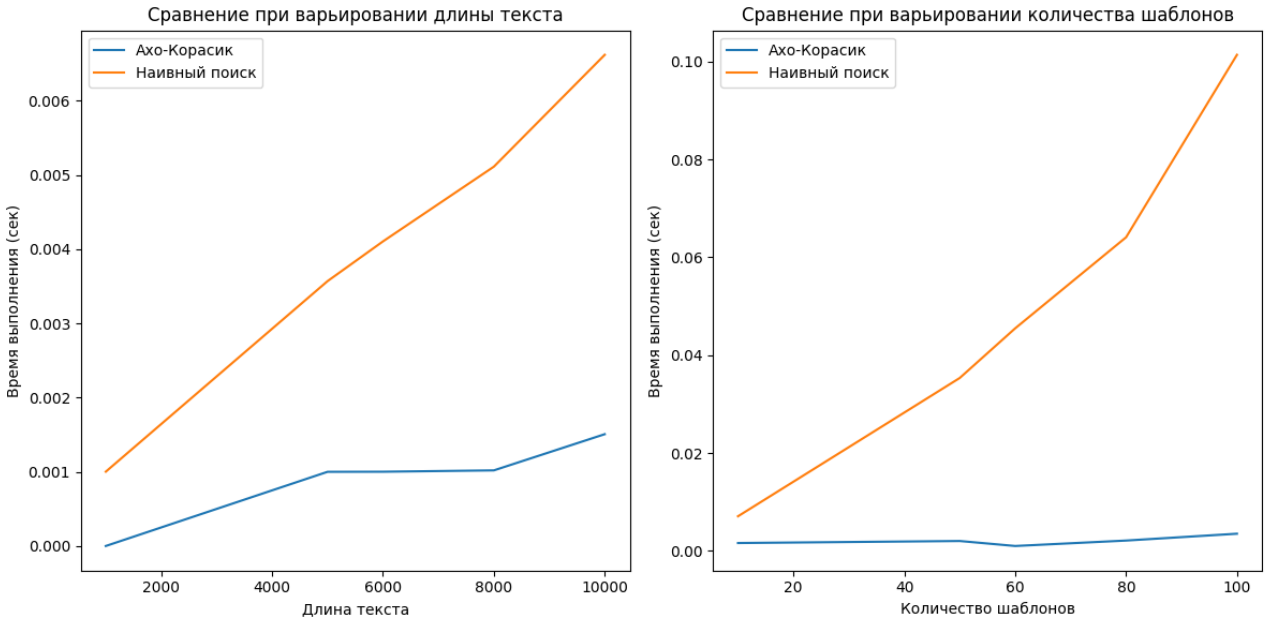

Рисунок 8 – Сравнение алгоритма Ахо-Корасик и наивного поиска

Можно сделать вывод, что Ахо-Корасик выполняется значительно быстрее, чем наивный алгоритм.

Выводы.

Изучен принцип работы алгоритма Ахо-Корасик. Написаны программы, корректно решающие задачу поиска набора подстрок в строке, в также программа поиска подстроки с джокером.

ПРИЛОЖЕНИЕА
ИСХОДНЫЙКОДПРОГРАММЫ

Название файла: index.ts

```
export type Pattern = [string, number]

export type SearchEntry = {
    pattern: Pattern,
    at: number
}

export class Node {
    constructor(
        public children: Record<string, Node> = {},
        public output: Pattern[] = [],
        public fail: Node | null = null,
    ) {}
}
export class SearchTree {
    protected patterns: Pattern[]
    public root: Node = new Node()

    public addPattern(pattern: Pattern) {
        let node = this.root

        for (const char of pattern[0]) {
            node.children[char] ??= new Node()
            node = node.children[char]
        }

        node.output.push(pattern);
    }

    private setLinks() {
        const queue: Node[] = []

        for (const char in this.root.children) {
            const child = this.root.children[char]

            child.fail = this.root;
            queue.push(child)

        }
```

```
    while (queue.length > 0) {
        const node = queue.shift()!

        for (const char in node.children) {
            const child = node.children[char]

            queue.push(child)

            let fail = node.fail
            while (fail && !(char in fail.children)) {
                fail = fail.fail
            }

            if (!fail) {
                child.fail = this.root
            } else {
                child.fail = fail.children[char]
            }

            child.output.push(...child.fail.output)
        }
    }
}

public search(text: string): SearchEntry[] {
    let node = this.root
    const results = []

    for(let i = 0; i < text.length; i++) {
        const char = text[i]

        while(!node.children[char] && node.fail) {
            node = node.fail
        }

        if(node.children[char]) {
            node = node.children[char]
        } else {
            node = this.root
        }

        for(const pattern of node.output) {
```

```
                results.push({    pattern:    pattern,    at:    i    -
pattern[0].length + 1 })
            }
        }

        return results
    }

    constructor(patterns: string[]) {
        this.patterns = patterns.map((p, i) => [p, i] as Pattern);
        for (const pattern of this.patterns) {
            this.addPattern(pattern);
        }
        this.setLinks()
    }
}
```

Название файла **wildcard.ts**

```
class WildcardSearchTree extends SearchTree {
    private positions: number[]

    constructor(
        private pattern: string,
        private wildcard: string,
    ) {
        const patterns = pattern.split(wildcard).filter(s => s)
        const positions = [];

        let flag = true;
        for(let i = 0; i < pattern.length; i++) {
            if(pattern[i] !== wildcard && flag) {
                positions.push(i)
                flag = false;
            }

            if(pattern[i] === wildcard) {
                flag = true;
            }
        }

        super(patterns)
        this.positions = positions
    }
```

```
    public search(text: string): SearchEntry[] {
        if(this.patterns.length === 0) return [];

        const fragmentMatches = super.search(text)
        const matches: Record<number, number[]> = {}

        for(const { pattern, at } of fragmentMatches) {
            const [ fragment, idx ] = pattern;
            const offset = this.positions[idx]
            const start = at - offset;

            if(start >= 0 && start <= text.length - this.pattern.length)
{
                matches[start] ??= [];
                matches[start].push(idx)
            }
        }

        const results: SearchEntry[] = []
        for(const                        position                        of
Object.keys(matches).toSorted().map(Number)) {
            const matched = new Set(matches[position])

            if(matched.size === this.patterns.length) {
                results.push({    pattern:    [this.pattern,    0],    at:
Number(position + 1) })
            }
        }

        return results
    }
}
```