Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчет по лабораторной работе №1 по дисциплине «Анализ алгоритмов»

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Алькина А.Р.

Группа ИУ7-54Б

Оценка (баллы) _____

Преподаватели Волкова Л. Л., Строганов Д. В.

Москва — 2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Расстояние Левенштейна — минимальное количество редакторских операций вставки, удаления и замены символа, которое необходимо для преобрзования одной строки в другую.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн [1] при изучении последовательностей нулей и единиц, впоследствии более общую задачу для произвольного алфавита связали с его именем. Большой вклад в изучение вопроса внёс Дэн Гасфилд.

Расстояние Левенштейна применяется в следующих областях.

1) Компьютерная лингвистика для исправления ошибок.

2) Сравнение текстовых файлов утилитой `diff` и другими.

3) Биоинформатика — для сравнения генов, хромосом и белков.

Расстояние Дамерау-Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира Левенштейна) — минимальное количество редакторских операций вставки, удаления, замены символа или транспозиции символов, которое необходимо для преобразования одной строки в другую.

Цель лабораторной работы — изучение, анализ и реализация алгоритмов нахождения расстояний между строками Левенштейна и Дамерау-Левенштейна.

Задачи лабораторной работы:

— изучить алгоритмы Левенштейна и Дамерау-Левенштейна;

— применить методы динамического программирования для реализации алгоритмов;

— получить практические навыки реализации алгоритмов Левенштейна и Дамерау-Левенштейна;

— провести сравнительный анализ на основе экспериментальных данных;

— подготовить отчет по лабораторной работе.

1 Аналитическая часть

Чтобы вычислить расстояние Левенштейна вводятся редакционные предписания — последовательность действий, необходимых для получения из первой строки второй кратчайшим образом. Обычно действия обозначаются так: D (англ. delete) — удалить, I (англ. insert) — вставить, R (replace) — заменить, M (match) — совпадение.

Цены операций могут зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов, отражая разную вероятность мутаций в биологии, разную вероятность разных ошибок при вводе текста и т. д. В общем случае:

- $w(a, b)$ — цена замены символа a на символ b;
- $w(\lambda, b)$ — цена вставки символа b;
- $w(a, \lambda)$ — цена удаления символа a.

Для решения задачи о редакционном расстоянии необходимо найти последовательность редакционных предписаний, минимизирующую суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при

- $w(a, a) = 0$;
- $w(a, b) = 1,\ a \neq b$;
- $w(\lambda, b) = 1$;
- $w(a, \lambda) = 1$.

1.1. Рекурсивный алгоритм нахождения расстояния Левенштейна

Существует проблема взаимного выравнивания строк.

Следующая таблица показывает не минимальное количество редакторских операций, получившееся в результате неудачного взаимного расположения строк.

Таблица 1.1 – Пример неоптимального выбора расположения строк

s1	у	в	л	е	ч	е	н	и	е		
s2	р	а	з	в	л	е	ч	е	н	и	е
Операции	R	R	R	R	R	M	R	R	R	I	I

Сумма цен операций равна 10.

Следующая таблица показывает, как оптимальный выбор взаимного расположения строк влияет на применяемые редакторские операции.

Таблица 1.2 – Пример оптимального выбора расположения строк

s1			у	в	л	е	ч	е	н	и	е
s2	р	а	з	в	л	е	ч	е	н	и	е
Операции	I	I	R	M	M	M	M	M	M	M	M

Сумма цен операций равна 3, что и является расстоянием Левенштейна для данных строк.

Проблема решается введением рекуррентной формулы.

Обозначим:

— L1 — длина S1 (первой строки);

— L2 — длина S2 (второй строки);

— S1[1...i] — подстрока S1 длиной i, начиная с первого символа;

— S2[1...j] — подстрока S2 длиной j, начиная с первого символа;

— S1[i] — i-й символ строки S1;

— S2[j] — j-й символ строки S2.

Также введём следующую функцию:

$$f(a,b) = \begin{cases} 0 & \text{если a = b,} \\ 1 & \text{иначе.} \end{cases} \tag{1.1}$$

Тогда расстояние Левенштейна между двумя строками S1 и S2 может быть вычислено по следующей формуле, где функция $D(S1[1...i], S2[1...j])$ определена как:

$$D(S1[1...i], S[1...j]) = \begin{cases} 0 & \text{i = 0, j = 0,} \\ j & \text{i = 0, j > 0,} \\ i & \text{i > 0, j = 0,} \\ \min\{ & \\ \quad D(S1[1...i], S2[1...j-1]) + 1 & \\ \quad D(S1[1...i-1], S2[1...j]) + 1 & \text{i > 0, j > 0} \\ \quad D(S1[1...i-1], S2[1...j-1]) + & \\ \quad + f(S1[i], S2[j]) & \\ \}. & \end{cases} \tag{1.2}$$

Рекурсивный алгоритм реализует формулу 1.2. Функция D составлена исходя следующих соображений:

— для перевода из пустой строки в пустую требуется ноль операций;

— для перевода из пустой строки в строку a требуется $|a|$ операций (вставка), где $|a|$ — длина строки a;

— для перевода из строки a в пустую требуется $|a|$ операций (удаление).

Для перевода из строки $S1$ в строку $S2$ требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность проведения любых двух операций можно поменять, порядок проведения операций не важен. Если $S1', S2'$ — строки $S1$ и $S2$ без последнего символа соответственно, то цена преобразования из строки $S1$ в строку $S2$ может быть выражена как:

— сумма цены преобразования строки $S1$ в $S2$ и цены проведения операции удаления (1), которая необходима для преобразования $S1'$ в $S1$;

— сумма цены преобразования строки $S1$ в $S2$ и цены проведения операции вставки (1), которая необходима для преобразования $S2'$ в $S2$;

— сумма цены преобразования из $S1'$ в $S2'$ и цены проведения операции замены (1), предполагая, что $S1$ и $S2$ оканчиваются разные символы;

— цена преобразования из $S1'$ в $S2'$ при условии, что $S1$ и $S2$ оканчиваются на один и тот же символ.

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.2. Матричный алгоритм нахождения расстояния Левенштейна

Альтернативным подходом для более эффективного вычисления Левенштейна является использование матрицы для сохранения промежуточных значений. Этот подход позволяет избежать повторных вычислений множества значений $D(S1[1...i], S2[1...j])$ при больших значениях i и j. Вместо этого алгоритм заполняет матрицу $A_{|S1|+1,|S2|+1}$ построчно значениями $D(S1[1...i], S2[1...j])$.

Таким образом, вычисление значения $D(S1[1...i], S2[1...j])$ зависит от значений, сохраненных в матрице A. Вместо повторного вычисления, алгоритм может просто обращаться к матрице для получения необходимого значения. Это значительно сокращает количество вычислений и делает алгоритм более эффективным.

Искомое расстояние при этом будет находиться в элементе матрицы $A[i][j]$.

1.3. Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Исходя из того, что 80% ошибок при наборе текста приходится на транспозицию (два соседних символа меняются местами), Фредерик Дамерау предложил ввести четвёртую редакторскую операцию — транспозицию (перестановку).

Обозначим:

— L1 — длина S1 (первой строки);

— L2 – длина S2 (второй строки);

— S1[1...i] — подстрока S1 длиной i, начиная с первого символа;

— S2[1...j] — подстрока S2 длиной j, начиная с первого символа;

— S1[i] — i-й символ строки S1;

— S2[j] — j-й символ строки S2.

Определим функцию f:

$$f(a,b) = \begin{cases} 0 & \text{если a = b,} \\ 1 & \text{иначе.} \end{cases} \tag{1.3}$$

Тогда расстояние Дамерау-Левенштейна между двумя строками S1 и S2 может быть вычислено по следующей формуле, где вычисляемая функция $D(S1[1...i], S2[1...j])$ определена как:

$$D(S1[1...i], S[1...j]) = \begin{cases} 0 & \text{i} = 0, \text{j} = 0, \\ j & \text{i} = 0, \text{j} > 0, \\ i & \text{i} > 0, \text{j} = 0, \\ \min\{ & \\ \quad D(S1[1...i-1], S2[1...j-1]) + 1 & \\ \quad D(S1[1...i], S2[1...j-1]) + 1 & \text{если (i} > 1, \text{j} > 1 \\ \quad D(S1[1...i-1], S2[1...j]) + 1 & \text{S1[i]} = \text{S2[j - 1]}, \\ \quad D(S1[1...i-1], S2[1...j-1]) + & \text{S1[i - 1]} = \text{S2[j])}, \\ \quad +f(S1[i], S2[j]) & \\ \}, & \\ \min\{ & \\ \quad D(S1[1...i], S2[1...j-1]) + 1 & \text{иначе} \\ \quad D(S1[1...i-1], S2[1...j]) + 1 & \\ \quad D(S1[1...i-1], S2[1...j-1]) + & \\ \quad +f(S1[i], S2[j]) & \\ \}. & \end{cases} \tag{1.4}$$

1.4. Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

Матричный алгоритм нахождения расстояния Дамерау-Левенштейна реализуется практически идентично матричному алгоритму Левенштейна. Но в данном алгоритме также задействуется элемент матрицы $A[i-2][j-2]$, если возможна транспозиция символов на текущем шаге.

1.5. Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с заполнением матрицы

Рекурсивную реализацию алгоритма Дамерау-Левенштейна можно ускорить, комбинируя рекуррентный и матричный подход.

Для более эффективного вычисления расстояния Левенштейна параллельно можно заполнять матрицу, которая будет использоваться для кеширования уже вычисленных результатов рекурсивных вызовов. В процессе заполнения матрицы рекурсивно вызываются только те вычисления, которые еще не были обработаны. Результаты нахождения расстояния сохраняются в матрице, и если уже обработанные данные встречаются снова, расстояние для них не находится и алгоритм переходит к следующему шагу.

Вывод

Алгоритмы Левенштейна и Дамерау-Левенштейна для вычисления расстояния между строками могут быть реализованы как рекурсивно, так и итерационно. Оба алгоритма определяют расстояние между двумя строками путем вычисления минимального количества операций (вставки, удаления и замены символов, а также транспозиции), необходимых для преобразования одной строки в другую.

2 Конструкторская часть

2.1. Схема алгоритма Левенштейна

На рисунке 2.1 приведена схема матричного алгоритма вычисления расстояния Левенштейна.

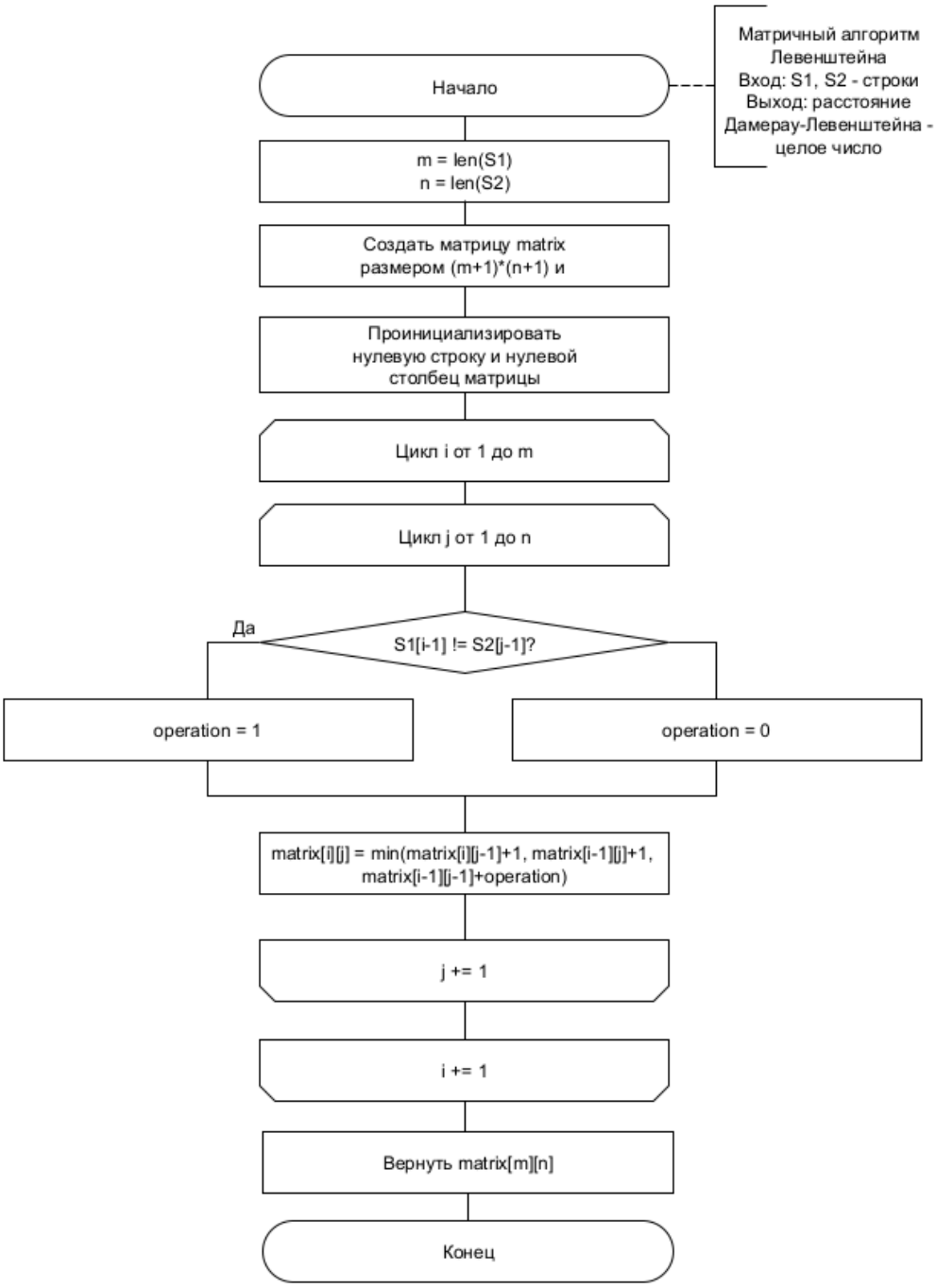

Рисунок 2.1 – Схема матричного алгоритма вычисления расстояния Левенштейна

2.2. Схема алгоритма Дамерау-Левенштейна

На рисунке 2.2 приведена схема рекурсивного алгоритма вычисления расстояния Дамерау-Левенштейна.

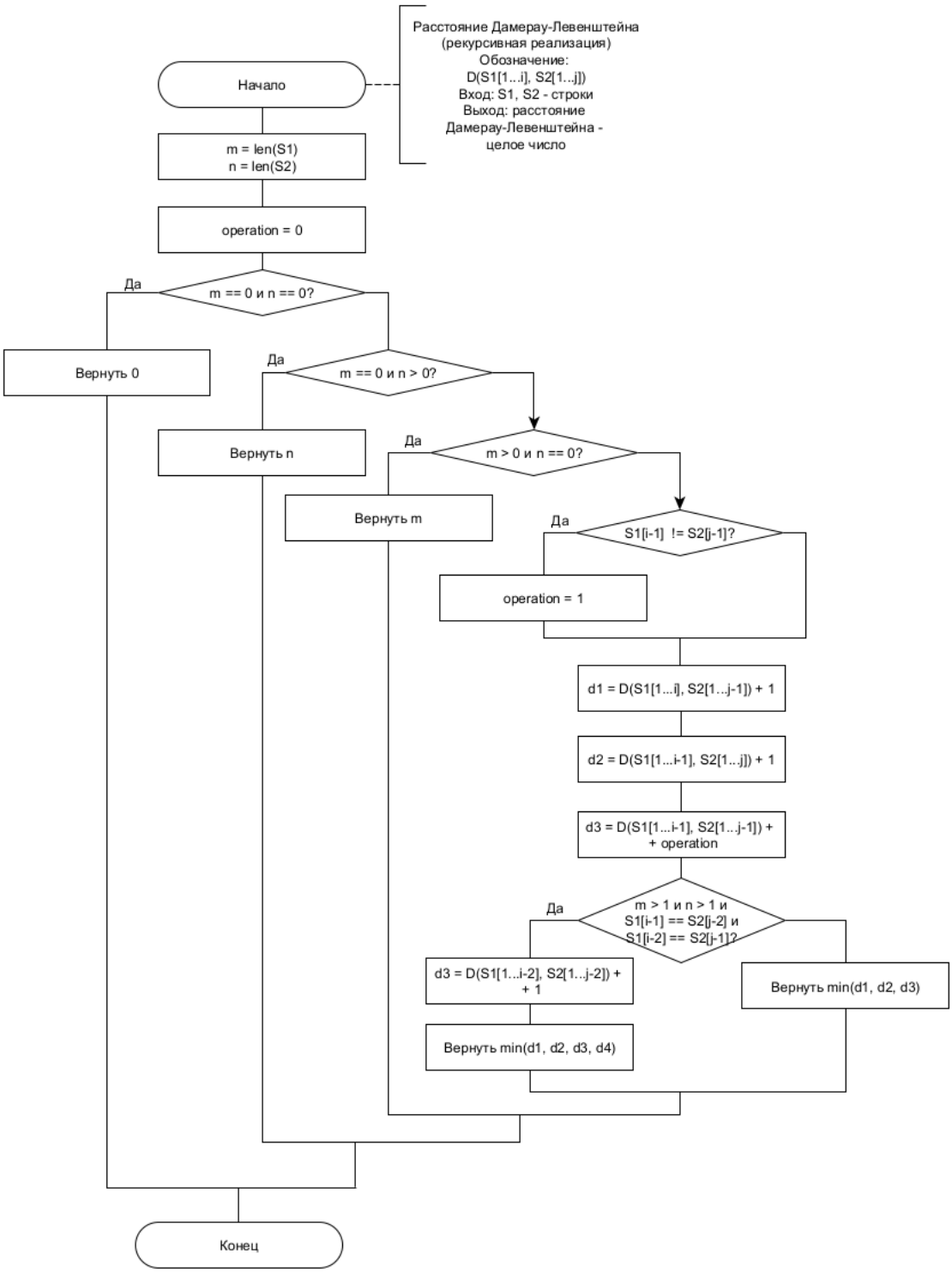

Рисунок 2.2 – Схема рекурсивного алгоритма вычисления расстояния Дамерау-Левенштейна

На рисунке 2.3 приведена схема рекурсивного алгоритма вычисления расстояния Дамерау-Левенштейна с заполнением матрицы. На дополнительном рисунке 2.4 приведена подпрограмма рекурсивного алгоритма вычисления расстояния Дамерау-Левенштейна с заполнением матрицы.

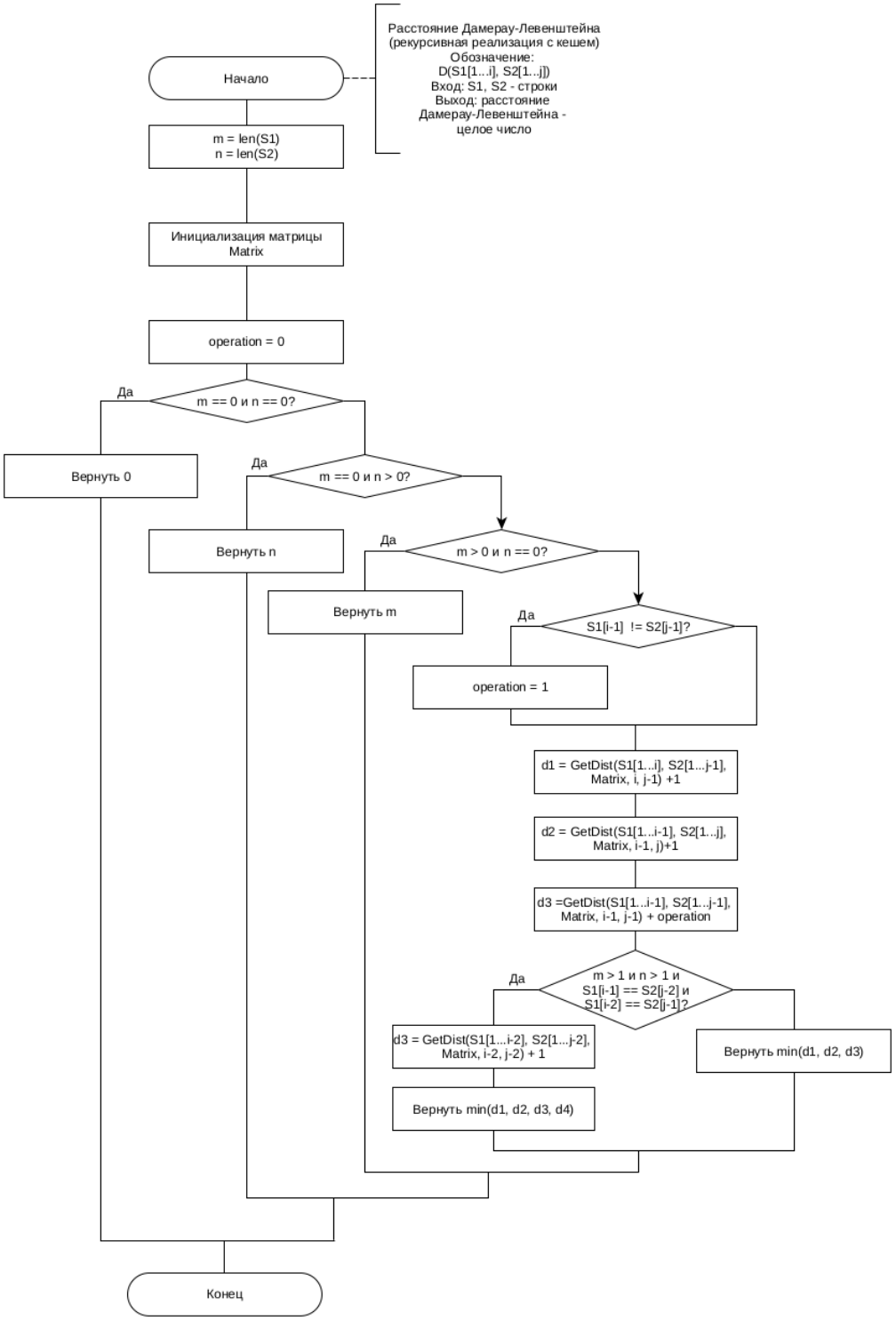

Рисунок 2.3 – Схема рекурсивного алгоритма Дамерау-Левенштейна с заполнением матрицы

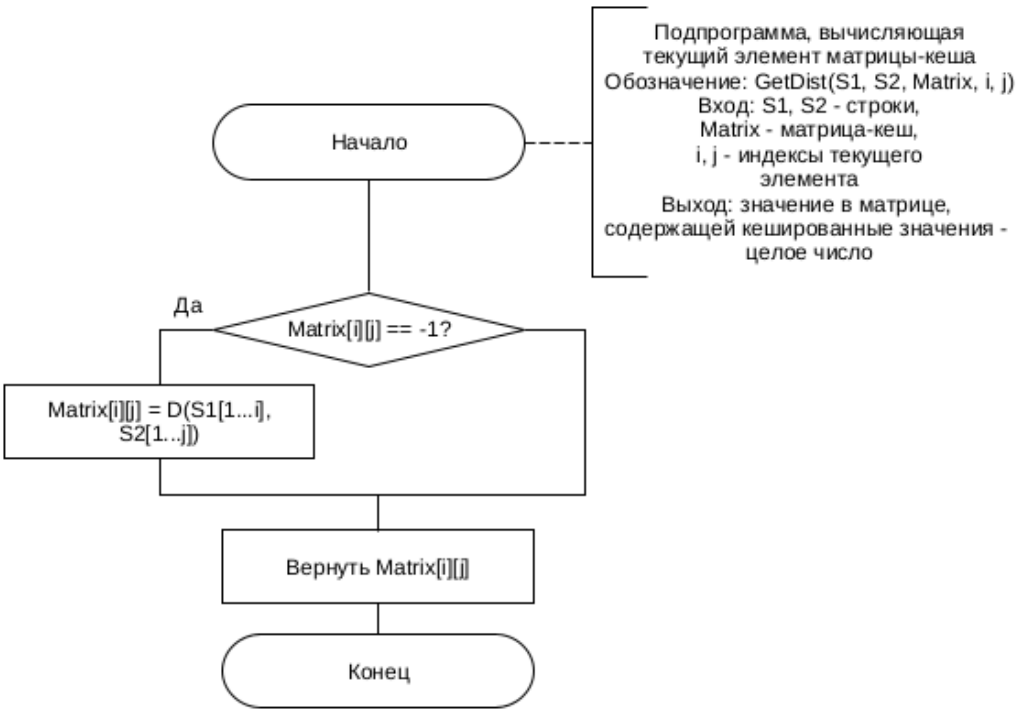

Рисунок 2.4 – Схема подпрограммы GetDist

На рисунке 2.5 приведена схема матричного алгоритма вычисления расстояния Дамерау-Левенштейна.

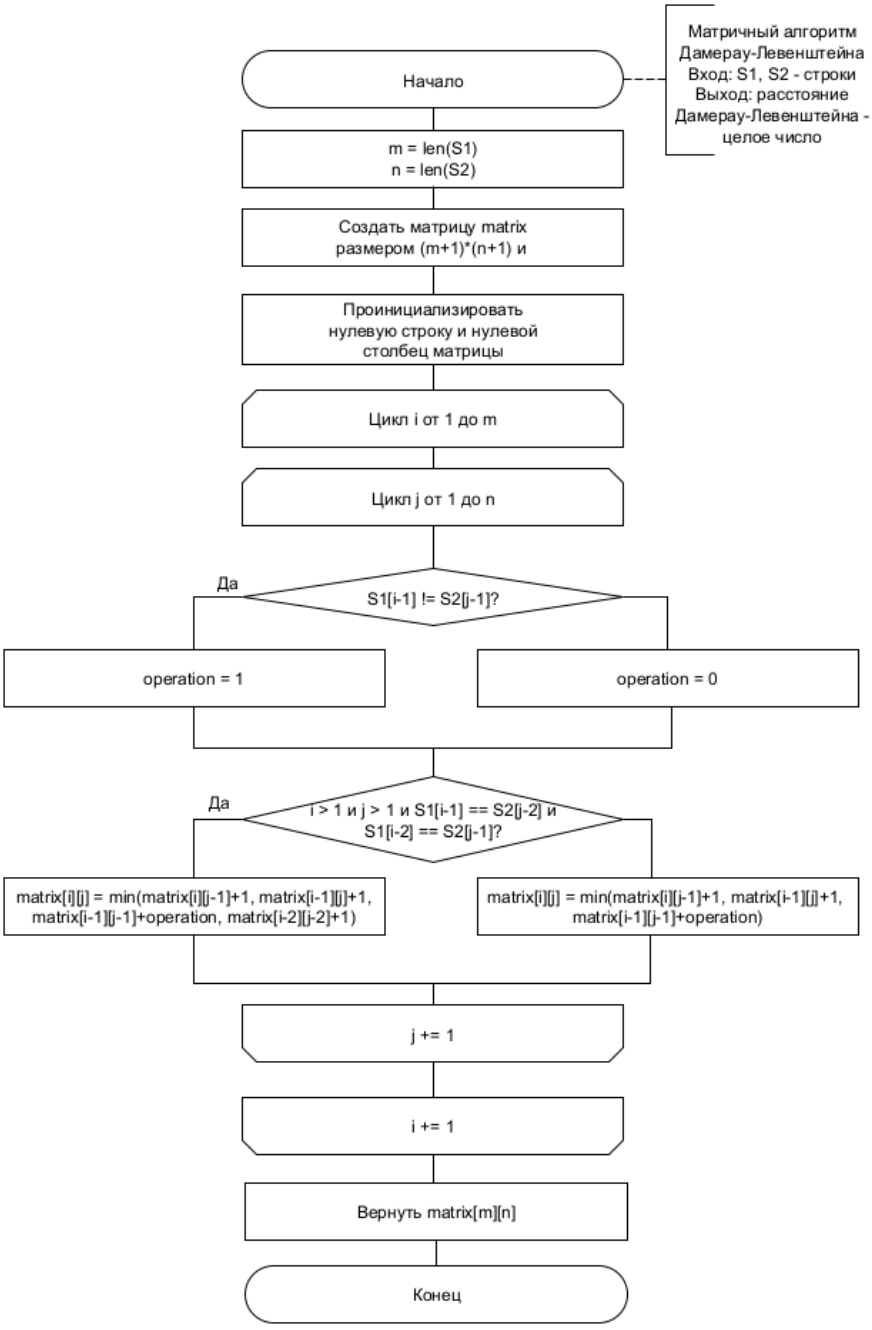

Рисунок 2.5 – Схема матричного алгоритма вычисления расстояния Дамерау-Левенштейна

Вывод

В данном разделе представлены схемы алгоритмов вычисления расстояний Левенштейна и Дамерау-Левенштейна между строками, построенные в соответствии с теоретическими сведениями из аналитического раздела.

3 Технологическая часть

В данном разделе представлены требования к программному обеспечению, а также рассматриваются средства реализации и приводятся листинги кода.

3.1. Требования к ПО

К программе предъявляется ряд требований:

— на вход подаются две строки;

— на выходе — искомое расстояние для всех четырех методов и матрицы расстояний для всех методов, которые подразумевают использование матрицы.

3.2. Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран многопоточный язык Go [2]. Выбор был сделан в пользу данного языка программирования, вследствие наличия пакетов для тестирования ПО.

3.3. Листинг кода

В листингах 3.1–3.4 приведены реализации алгоритмов Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Матричный (Левенштейн)

```
func MatrixLeven(str1 []rune, str2 []rune) (int, matrix.Matrix) {
    operation := 0
    matr := matrix.CreateMatrix(len(str1)+1, len(str2)+1, false)

    for i := 1; i < matr.M; i++ {
        for j := 1; j < matr.N; j++ {
            if str1[i-1] != str2[j-1] {
                operation = 1
            } else {
                operation = 0
            }
            matr.Matr[i][j] = GetMin(matr.Matr[i][j-1]+1, matr.Matr[i-1][j]+1,
                matr.Matr[i-1][j-1]+operation)
        }
    }

    return matr.Matr[matr.M-1][matr.N-1], matr
}
```

Листинг 3.2 – Рекурсивный (Дамерау-Левенштейн)

```
func DamerauLeven(str1, str2 []rune) int {
    m, n := len(str1), len(str2)
    var operation, transposition int

    if m == 0 && n == 0 {
        return 0
    } else if m == 0 && n > 0 {
        return n
    } else if m > 0 && n == 0 {
        return m
    } else {
        if str1[m-1] != str2[n-1] {
            operation = 1
        }
        moveLeft := DamerauLeven(str1, str2[:n-1]) + 1
        moveRight := DamerauLeven(str1[:m-1], str2) + 1
        moveDiag := DamerauLeven(str1[:m-1], str2[:n-1]) + operation
        if m >= 2 && n >= 2 && str1[m-1] == str2[n-2] && str1[m-2] == str2[n-1] {
            transposition = DamerauLeven(str1[:m-2], str2[:n-2]) + 1
            return GetMin(moveLeft, moveRight, moveDiag, transposition)
        }
        return GetMin(moveDiag, moveRight, moveLeft)
    }

    return 0
}
```

Листинг 3.3 – Матричный (Дамерау-Левенштейн)

```
func MatrixDamerauLeven(str1 []rune, str2 []rune) (int, matrix.Matrix) {
    operation := 0
    matr := matrix.CreateMatrix(len(str1)+1, len(str2)+1, false)

    for i := 1; i < matr.M; i++ {
        for j := 1; j < matr.N; j++ {
            if str1[i-1] != str2[j-1] {
                operation = 1
            } else {
                operation = 0
            }
            if i >= 2 && j >= 2 && str1[i-1] == str2[j-2] && str1[i-2] == str2[j-1] {
                matr.Matr[i][j] = GetMin(matr.Matr[i][j-1]+1, matr.Matr[i-1][j]+1,
                    matr.Matr[i-1][j-1]+operation, matr.Matr[i-2][j-2]+1)
            } else {
                matr.Matr[i][j] = GetMin(matr.Matr[i][j-1]+1, matr.Matr[i-1][j]+1,
                    matr.Matr[i-1][j-1]+operation)
            }
        }
    }

    return matr.Matr[matr.M-1][matr.N-1], matr
}
```

Листинг 3.4 – Рекурсивный с кешем (Дамерау-Левенштейн)

```
func GetDistance(matr *matrix.Matrix, i int, j int, str1 []rune, str2 []rune) int {
    if matr.Matr[i][j] == -1 {
        matr.Matr[i][j] = RecursivePartOfLeven(str1, str2, matr)
    }
    return matr.Matr[i][j]
}

func RecursivePartOfLeven(str1 []rune, str2 []rune, matr *matrix.Matrix) int {
    var operation int

    m, n := len(str1), len(str2)

    if m == 0 && n == 0 {
        return 0
    } else if m == 0 && n > 0 {
        return n
    } else if m > 0 && n == 0 {
        return m
    } else {
        if str1[m-1] != str2[n-1] {
            operation = 1
        }
        moveLeft := GetDistance(matr, m, n-1, str1, str2[:n-1]) + 1
        moveRight := GetDistance(matr, m-1, n, str1[:m-1], str2) + 1
        moveDiag := GetDistance(matr, m-1, n-1, str1[:m-1], str2[:n-1]) + operation
        if m >= 2 && n >= 2 && str1[m-1] == str2[n-2] && str1[m-2] == str2[n-1] {
            transposition := GetDistance(matr, m-2, n-2, str1[:m-2], str2[:n-2]) + 1
            return GetMin(moveLeft, moveRight, moveDiag, transposition)
        }
        return GetMin(moveLeft, moveRight, moveDiag)
    }
}

func RecursiveDamerauLevenWithCache(str1 []rune, str2 []rune) (int, matrix.Matrix) {
    matr := matrix.CreateMatrix(len(str1)+1, len(str2)+1, true)
    ans := RecursivePartOfLeven(str1, str2, &matr)
    matr.Matr[matr.M-1][matr.N-1] = ans

    return ans, matr
}
```

Листинги со служебным кодом находятся в приложении.

3.4. Функциональное тестирование

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

Обозначим:

— 1 — алгоритм вычисления расстояния Левенштейна матричный;

— 2 — алгоритм вычисления расстояния Дамерау-Левештейна рекурсивный;

— 3 — алгоритм вычисления расстояния Дамерау-Левенштейна матричный;

— 4 — алгоритм вычисления расстояния Дамерау-Левенштейна с кешем.

Таблица 3.1 – Функциональные тесты

Строка 1	**Строка 2**	**Ожидаемый результат**			
		1	**2**	**3**	**4**
equal	equal	0	0	0	0
call	code	2	2	2	2
program	prorgam	2	1	1	1
program	programmer	3	3	3	3

3.5. Пример работы

Демонстрация работы программы приведена на рисунке 3.1.

```
Выберите опцию: 1
Введите строку 1: one
Введите строку 2: two

Выберите опцию: 3

Редакционное расстояние между строками = 3
    0     1     2     3
    1     1     2     2
    2     2     2     3
    3     3     3     3


Выберите опцию: 4

Редакционное расстояние между строками = 3

Выберите опцию: 5

Редакционное расстояние между строками = 3
Матрица:

    0     1     2     3
    1     1     2     2
    2     2     2     3
    3     3     3     3


Выберите опцию: 6

Редакционное расстояние между строками = 3
Матрица:

    0     1     2     3
    1     1     2     2
    2     2     2     3
    3     3     3     3

Выберите опцию: 0
```

Рисунок 3.1 – Демонстрация работы программы

Вывод

В результате, были разработаны следующие алгоритмы: алгоритм нахождения редакционного расстояния Левенштейна, алгоритм нахождения редакционного расстояния Дамерау-Левенштейна, алгоритм вычисления редакционного расстояния Дамерау-Левенштейна итерационный, алгоритм вычисления редакционного расстояния Дамерау-Левенштейна с кешированием.

4 Исследовательская часть

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

— операционная система: Ubuntu [3] 22.04.3 LTS;

— оперативная память: 15 ГБ;

— процессор: 12 ядер, AMD Ryzen 5 4600ГЦ with Radeon Graphics [4].

4.2. Время выполнения алгоритмов

Во время замеров времени система не была нагружена дополнительными пользовательскими процессами.

Замеры проводились с помощью разработанной функции GetCPU, которая использует модуль syscall [5]. syscall.Rusage является структурой в пакете syscall, предназначенной для хранения информации о ресурсах, используемых процессом или потоком. Эта структура содержит различные поля, такие как Utime (время использования ЦПУ в пользовательском режиме), Stime (время использования ЦПУ в режиме ядра) и так далее.

Для получения времени использования ЦПУ, извлекаем атрибуты структуры Utime и Stime.

Выражение usage.Utime.Nano() + usage.Stime.Nano() возвращает сумму времени использования ЦПУ в наносекундах для процесса или потока. Это значение может быть использовано для измерения общего времени использования ЦПУ в пределах процесса или потока.

Результаты замеров приведены в таблице 4.1.

Обозначим:

— 1 — матричный алгоритм вычисления расстояния Левенштейна между строками;

— 2 — матричный алгоритм вычисления расстояния между строками Дамерау-Левештейна;

— 3 — рекурсивный алгоритм вычисления расстояния между строками Дамерау-Левенштейна;

— 4 — рекурсивный алгоритм с кешем вычисления расстояния Дамерау-Левенштейна.

Таблица 4.1 – Замер времени для строк, размером от 5 до 300

Длина строк	**Время, нс**			
	1	**2**	**3**	**4**
5	1936	657	17134	951
10	5421	1265	38603216	2478
15	25600	3200	213888700000	7600
50	64653	59871	None	173452
100	235207	233723	None	586421
200	882743	855473	None	2526968
300	2078467	2104723	None	5863640

На следующем графике представлена зависимость времени работы алгоритма вычисления расстояния Левенштейна и Дамерау-Левенштейна от длины строк (до 15 символов).

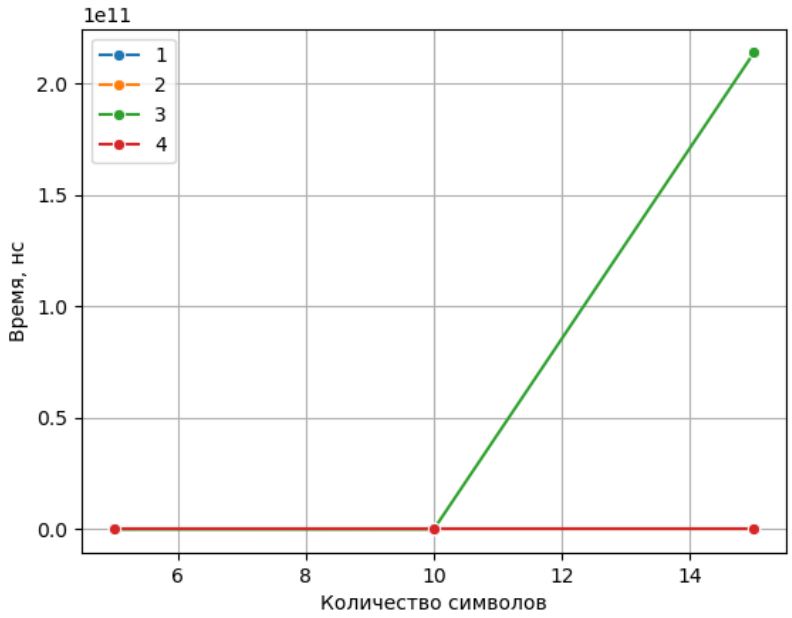

Рисунок 4.1 – Время работы алгоритмов вычисления расстояния между строками

На следующем графике представлена зависимость времени работы алгоритма вычисления расстояния Левенштейна и Дамерау-Левенштейна от длины строк (до 300 символов, без рекурсивной реализации алгоритма Дамерау-Левенштейна).

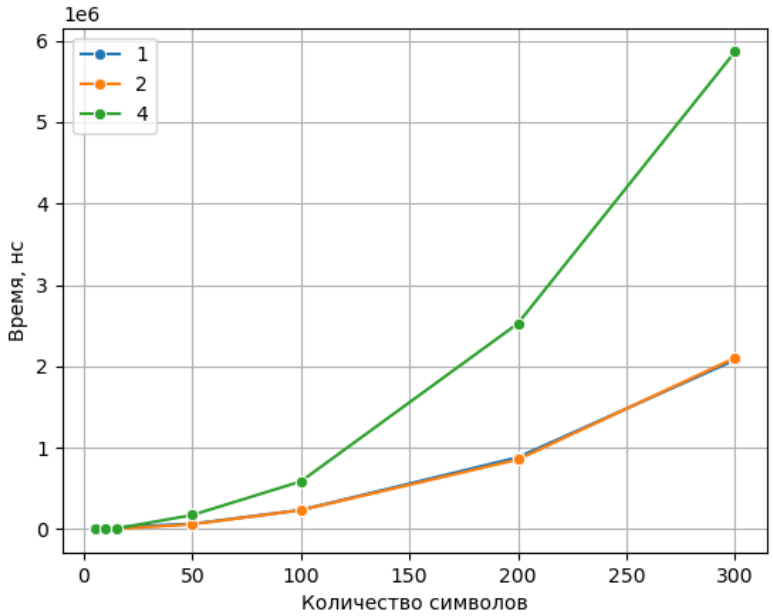

Рисунок 4.2 – Время работы алгоритмов вычисления расстояния между строками

4.3. Использование памяти

Алгоритмы, использующие матрицы, не отличаются друг от друга по потреблению памяти, поэтому ниже приведено сравнение только рекурсивной и матричной реализации.

Матричный алгоритм Дамерау-Левенштейна:

$$11 \cdot len(\text{int}) + len(\text{rune}) \cdot (len(\text{S}_1) + len(\text{S}_2)) + len(\text{Matrix}), \tag{4.1}$$

где

$$len(\text{Matrix}) = (len(\text{S}_1) + len(\text{S}_2) + 2) \cdot len(\text{int}) + 2 \cdot len(\text{int}) + len(\text{bool}), \tag{4.2}$$

где len — оператор вычисления размера, S_1, S_2 — строки, int — целочисленный тип, rune — строковый тип в Go, Matrix — пользовательский тип данных, представляющий матрицу, bool — логический тип данных.

Рекурсивный алгоритм Дамерау-Левенштейна: Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти

$$(len(\text{S}_1) + len(\text{S}_2)) \cdot (2 \cdot len(\text{rune}) + 15 \cdot len(\text{int})), \tag{4.3}$$

где len — оператор вычисления размера, S_1, S_2 — строки, int — целочисленный тип, rune — строковый тип в Go.

Выделение памяти при работе алгоритмов указано на рисунке 4.1. Получено при помощи команды: `go test -bench . -benchmem`

Листинг 4.1 – Матричный (Левенштейн)

```
goos: linux
goarch: amd64
pkg: lab1.com/algs1
cpu: AMD Ryzen 5 4600H with Radeon Graphics
BenchmarkDamerauLeven5-12 81226 13294 ns/op 0 B/op 0 allocs/op
BenchmarkDamerauLen10-12 18 65501505 ns/op 0 B/op 0 allocs/op
BenchmarkDamerauLevenLen15-12 1 212854803869 ns/op 0 B/op 0 allocs/op
BenchmarkMatrixDamerauLevenLen5-12 1000000 1042 ns/op 432 B/op 7 allocs/op
BenchmarkMatrixDamerauLevenLen10-12 369198 2868 ns/op 1344 B/op 12 allocs/op
BenchmarkMatrixDamerauLevenLen15-12 197188 5785 ns/op 2432 B/op 17 allocs/op
BenchmarkMatrixDamerauLevenLen50-12 22540 52393 ns/op 22496 B/op 52 allocs/op
BenchmarkMatrixDamerauLevenLen100-12 5368 206314 ns/op 93184 B/op 102 allocs/op
BenchmarkMatrixDamerauLevenCacheLen5-12 703873 1915 ns/op 432 B/op 7 allocs/op
BenchmarkMatrixDamerauLevenCacheLen10-12 277136 6075 ns/op 1344 B/op 12 allocs/op
BenchmarkMatrixDamerauLevenCacheLen15-12 98534 12546 ns/op 2432 B/op 17 allocs/op
BenchmarkMatrixDamerauLevenCacheLen50-12 7834 156253 ns/op 22496 B/op 52 allocs/op
BenchmarkMatrixDamerauLevenCacheLen100-12 1585 731190 ns/op 93184 B/op 102 allocs/op
BenchmarkMatrixLevenLen5-12 1000000 1038 ns/op 432 B/op 7 allocs/op
BenchmarkMatrixLevenLen10-12 411806 2838 ns/op 1344 B/op 12 allocs/op
BenchmarkMatrixLevenLen15-12 210435 5360 ns/op 2432 B/op 17 allocs/op
BenchmarkMatrixLevenLen50-12 23905 49641 ns/op 22496 B/op 52 allocs/op
BenchmarkMatrixLevenLen100-12 5313 202286 ns/op 93184 B/op 102 allocs/op
PASS
ok lab1.com/algs1 237.469s
```

Вывод

Рекурсивный алгоритм Дамерау-Левенштейна имеет большую вычислительную сложность и работает медленнее итеративных реализаций. Время выполнения рекурсивного алгоритма увеличивается экспоненциально с ростом длины строк. Например, на словах длиной 10 символов, матричная реализация алгоритма Дамерау-Левенштейна работает минимум в 12000 раз быстрее, чем рекурсивная реализация.

Однако, рекурсивный Дамерау-Левенштейна имеет преимущество в использовании памяти. Итеративные алгоритмы требуют большего объема памяти, так как их потребление памяти растет как произведение длин строк. В то же время, рекурсивный алгоритм требует памяти пропорционально сумме длин строк.

Алгоритм Дамерау-Левенштейна по времени выполнения сопоставим с алгоритмом Левенштейна.

ЗАКЛЮЧЕНИЕ

В ходе выполнения работы были выполнены все поставленные задачи, в том числе изучение методов динамического программирования на основе алгоритмов вычисления расстояния Левенштейна.

Исследование позволило выявить различия в производительности различных алгоритмов вычисления расстояния Левенштейна. В частности, матричные алгоритмы продемонстрировали большее потребление памяти по сравнению с рекурсивными из-за выделения дополнительной памяти под матрицы и использования большего количества локальных переменных.

Также были проведены теоретические расчеты использования памяти в каждом из алгоритмов вычисления расстояния Левенштейна, включая алгоритм Дамерау-Левенштейна. В результате рекурсивный алгоритм продемонстрировал более эффективное использование памяти по сравнению с матричными алгоритмами, которые требуют дополнительного выделения памяти под матрицы и более широкий набор локальных переменных.

В целом, исследование позволило получить практические и теоретические результаты, подтверждающие различия в производительности и использовании памяти различных алгоритмов вычисления расстояния Левенштейна. Эти результаты могут служить основой для выбора наиболее эффективного алгоритма в зависимости от конкретных требований и ограничений проекта.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 15.09.2023).

[3] Linux — Официальная документация [Электронный ресурс]. Режим доступа: `https://www.linux.org/` (дата обращения: 15.09.2023).

[4] AMD Processors [Электронный ресурс]. Режим доступа: `https://www.amd.com/en.html` (дата обращения: 15.09.2023).

[5] Пакет syscall [Электронный ресурс]. Режим доступа: `https://pkg.go.dev/syscall` (дата обращения: 15.09.2023).

ПРИЛОЖЕНИЕ 1

В листинге 4.2 приведена структура Matrix.

Листинг 4.2 – Структура Matrix

```
package matrix

import (
    "fmt"
)

type Matrix struct {
    Matr [][]int
    M int
    N int
    Inf bool
}

func (m *Matrix) MakeMatrix() {
    m.Matr = make([][]int, m.M)
    for i := range m.Matr {
        m.Matr[i] = make([]int, m.N)
    }
}

func (m *Matrix) InitMatrix() {
    for j := 0; j < m.N; j++ {
        m.Matr[0][j] = j
    }

    for i := 0; i < m.M; i++ {
        m.Matr[i][0] = i
    }

    if m.Inf {
        for i := 1; i < m.M; i++ {
            for j := 1; j < m.N; j++ {
                m.Matr[i][j] = -1
            }
        }
    }
}

func (m *Matrix) OutputMatrix() {
    for i := 0; i < m.M; i++ {
        for j := 0; j < m.N; j++ {
            fmt.Printf("%5d␣", m.Matr[i][j])
        }
```

```
        fmt.Println()
    }
    fmt.Println()
}

func CreateMatrix(m, n int, inf bool) Matrix {
    matr := Matrix{M: m, N: n, Inf: inf}
    matr.MakeMatrix()
    matr.InitMatrix()
    return matr
}
```

В листинге 4.3 приведен модуль для замеров процессорного времени выполнения программы.

Листинг 4.3 – Измерение времени

```
package time_measure

import (
    "math/rand"
    "syscall"

    "lab1.com/algs1"
    "lab1.com/matrix"
)

const N int = 1000
const symbols string = "abcdefghijklmnopqrstuvwxyz"

const maxInd int = 26

func GetCPU() int64 {
    usage := new(syscall.Rusage)
    syscall.Getrusage(syscall.RUSAGE_SELF, usage)
    return usage.Utime.Nano() + usage.Stime.Nano()
}

func GetRandomString(len int) []rune {
    rstr := make([]rune, len)
    for i := 0; i < len; i++ {
        symb := rand.Intn(maxInd)
        rstr[i] = rune(symbols[symb])
    }
    return rstr
}

func MatrixLevenTimeMeasurement(str1 []rune, str2 []rune) (float32, int) {
    var sum float32
```

```
33
34      var startTime, finishTime int64
35      var ans int
36      var mtr matrix.Matrix
37
38      for i := 0; i < N; i++ {
39          startTime = GetCPU()
40          ans, mtr = algs1.MatrixLeven(str1, str2)
41          finishTime = GetCPU()
42          sum += float32(finishTime - startTime)
43      }
44
45      mtr.Matr[0][0] += 1
46
47      return (sum / float32(N)) / 1, ans
48  }
49
50  func DamerauLevenTimeMeasurement(str1 []rune, str2 []rune) (float32, int) {
51      var sum float32
52
53      var startTime, finishTime int64
54      var ans int
55
56      for i := 0; i < N; i++ {
57          startTime = GetCPU()
58          ans = algs1.DamerauLeven(str1, str2)
59          finishTime = GetCPU()
60          sum += float32(finishTime - startTime)
61      }
62
63      return (sum / float32(N)) / 1, ans
64  }
65
66  func MatrixDamerauLevenTimeMeasurement(str1 []rune, str2 []rune) (float32, int) {
67      var sum float32
68
69      var startTime, finishTime int64
70      var ans int
71      var matr matrix.Matrix
72
73      for i := 0; i < N; i++ {
74          startTime = GetCPU()
75          ans, matr = algs1.MatrixDamerauLeven(str1, str2)
76          finishTime = GetCPU()
77          sum += float32(finishTime - startTime)
78      }
79
80      matr.Matr[0][0] += 1
```

```
81
82      return (sum / float32(N)) / 1, ans
83  }
84
85  func RecursiveDamerauLevenWithCacheTimeMeasurement(str1 []rune, str2 []rune) (float32,
        int) {
86      var sum float32
87
88      var startTime, finishTime int64
89      var ans int
90      var matr matrix.Matrix
91
92      for i := 0; i < N; i++ {
93          startTime = GetCPU()
94          ans, matr = algs1.RecursiveDamerauLevenWithCache(str1, str2)
95          finishTime = GetCPU()
96          sum += float32(finishTime - startTime)
97      }
98
99      matr.Matr[0][0] += 1
100
101     return (sum / float32(N)) / 1, ans
102 }
103
104 func MeasureTime(rstr1 []rune, rstr2 []rune) [4]float32 {
105     var seconds [4]float32
106     var ans int
107
108     seconds[0], ans = MatrixLevenTimeMeasurement(rstr1, rstr2)
109     seconds[1], ans = DamerauLevenTimeMeasurement(rstr1, rstr2)
110     seconds[2], ans = MatrixDamerauLevenTimeMeasurement(rstr1, rstr2)
111     seconds[3], ans = RecursiveDamerauLevenWithCacheTimeMeasurement(rstr1, rstr2)
112
113     ans += 1
114
115     return seconds
116 }
```