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»

Отчёт по лабораторной работе №1 по курсу "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Леонов В.В.

Группа ИУ7-56Б

Оценка (баллы) ______________

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Содержание

Введение **3**

1 Аналитическая часть **5**

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна 6

1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшированием 7

1.3 Расстояния Дамерау — Левенштейна 8

1.4 Матричный алгоритм расстояния Дамерау–Левенштейна . . 8

1.5 Вывод 9

2 Конструкторская часть **10**

2.1 Схема алгоритма Левенштейна 10

2.2 Схема алгоритма Дамерау — Левенштейна 12

2.3 Вывод 13

3 Технологическая часть **14**

3.1 Требования к ПО 14

3.2 Средства реализации 14

3.3 Листинг кода 14

3.4 Вывод 19

4 Исследовательская часть **20**

4.1 Пример работы 20

4.2 Технические характеристики 21

4.3 Время выполнения алгоритмов 21

4.4 Использование памяти 21

4.5 Вывод 22

Заключение **25**

Список литературы **26**

Введение

Современные компьютеры являются мощными устройствами для работы с текстом, они осуществляют его хранение, передачу и обработку. Согласно статистике каждую минуту происходит создание порядка 16 миллионов текстовых сообщений, больше половины человечества используют электронную почту, отправляя ежедневно 267 миллиардов электронных писем, и число пользователей только увеличивается, так к 2023 году ожидается их рост до 5.3 миллиарда. При этом почти 60 % электронных посланий содержат опечатки или написаны с ошибками.

Автоматическая проверка орфографии — одна из актуальных проблем в области обработки естественного языка, универсального решения для нее до сих пор не представлено, однако на протяжении всей своей истории коррекция орфографии являлась актуальной задачей прикладной лингвистики. Таким образом, существует задача определения максимально похожего слова к написанному, которую в свою очередь описывает расстояние Левенштейна.

Данная метрика и ее вариации активно и часто используются:

1) для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

2) в биоинформатике для сравнения генов, хромосом и белков.

Расстояние Левенштейна [1] — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены.

Расстояние Дамерау — Левенштейна яляется модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Целью данной лабораторной работы являются изучение и реализация алгоритмов Левенштейна и Дамерау–Левенштейна.

Для достижения указанной выше цели следует выполнить следующие задачи:

- изучение алгоритмов Левенштейна и Дамерау–Левенштейна;
- привести схемы указанных алгоритмов поиска редакционного расстояния;
- применение методов динамического программирования для реализации указанных алгоритмов;
- выполнение сравнительного анализа линейной и рекурсивной реализаций алгоритмов по затрачиваемым ресурсам (по памяти и по времени);
- описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1 Аналитическая часть

Расстояние Левенштейна - минимальное количество операций вставки/удаления одного символа, а также замены одного символа на другой, необходимых для преобразования одной строки в другую.

Расстояние Дамерау–Левенштейна вычисляется аналогично, с учетом добавления часто применяющейся операции, которую заметил Дамерау: транспозиция 2 соседних символов в строке.

Задача по поиску расстояния Левенштейна и Дамерау–Левенштейна соотвественно заключается в нахождении последовательности этих операций, стоимость которых будет минимальной.

Задаются базовые редакторские операции, т.е. правила, а также вводится понятие штрафа - цена одной операции преобразования строки.

Базовые операции:

1. **I(Insert)** - вставка символа в строку.
   Штраф = 1

2. **D(Delete)** - удаление символа из строки.
   Штраф = 1

3. **R(Replace)** - замена символа в строке.
   Штраф = 1

4. **M(Match)** - совпадение символа из первой строки с символом из второй строки.
   Штраф = 0

5. **X(eXchange)** - транспозиция двух соседних символов в строке.
   Штраф = 1

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле 1.1, где $|a|$ означает длину строки a; $a[i]$ — i-ый символ строки a , функция $D(i, j)$ определена как:

$$D(i,j) = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{j = 0, i > 0} \\ j & \text{i = 0, j > 0} \\ \min\{ & \\ \qquad D(i, j-1) + 1 & \\ \qquad D(i-1, j) + 1 & \text{i > 0, j > 0} \\ \qquad D(i-1, j-1) + m(a[i], b[j]) & (1.2) \\ \} & \end{cases}, \tag{1.1}$$

а функция 1.2 определена как:

$$m(a, b) = \begin{cases} 0 & \text{если a[i] = b[j],} \\ 1 & \text{иначе} \end{cases}. \tag{1.2}$$

Рекурсивный алгоритм реализует формулу 1.1. Функция D составлена из следующих соображений:

1) для перевода из пустой строки в пустую требуется ноль операций;

2) для перевода из пустой строки в строку a требуется $|a|$ операций;

3) для перевода из строки a в пустую требуется $|a|$ операций.

Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность проведения любых двух операций можно поменять, порядок проведения операций не имеет никакого значения. Полагая, что a', b' — строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

1) сумма цены преобразования строки a в b и цены проведения операции удаления, которая необходима для преобразования a' в a;

2) сумма цены преобразования строки a в b и цены проведения операции вставки, которая необходима для преобразования b' в b;

3) сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются разные символы;

4) цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшированием

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием матричного алгоритма. Суть данного метода заключается в параллельном заполнении матрицы при выполнении рекурсии. В случае, если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, результат нахождения расстояния заносится в матрицу. В случае, если обработанные ранее данные встречаются снова, для них расстояние не находится и алгоритм переходит к следующему шагу.

1.3 Расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна может быть найдено по формуле 1.3, которая задана как

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1)+1, & \\ \quad d_{a,b}(i-1,j)+1, & \text{иначе} \\ \quad d_{a,b}(i-1,j-1)+m(a[i],b[j]), & \\ \quad \left[\begin{array}{ll} d_{a,b}(i-2,j-2)+1, & \text{если } i,j>1; \\ & a[i]=b[j-1]; \\ & b[j]=a[i-1] \\ \infty, & \text{иначе} \end{array} \right. & \\ \} & \end{cases}, \tag{1.3}$$

Формула выводится по тем же соображениям, что и формула (1.1).

1.4 Матричный алгоритм расстояния Дамерау–Левенштейна

Прямая реализация формулы 1.3 может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений $D(i, j)$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Дамерау–Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{|a|,|b|}$ значениями $d(i, j)$.

1.5 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау–Левенштейна, задача которых состоит в том, чтобы определить минимальное количество операций вставки/удаления одного символа, а также замены одного символа на другой и транспозиции двух пар соседних символов, необходимых для преобразования одной строки в другую.

Формулы для вычисления задаются в рекурсивном виде (см. формулы 1.3 и 1.1). Как известно, рекурсия - часто не самый эффективный способ решения, на больших данных будет затрачиваться большое количество памяти и времени, поэтому был рассмотрен способ оптимизации вычислений, в частности использование матрицы для хранения промежуточных ответов.

2 Конструкторская часть

В данном разделе представлены схемы алгоритмов, описанных в аналитическом разделе.

2.1 Схема алгоритма Левенштейна

На рисунке 2.1 приведена схема рекурсивного алгоритма Левенштейна.

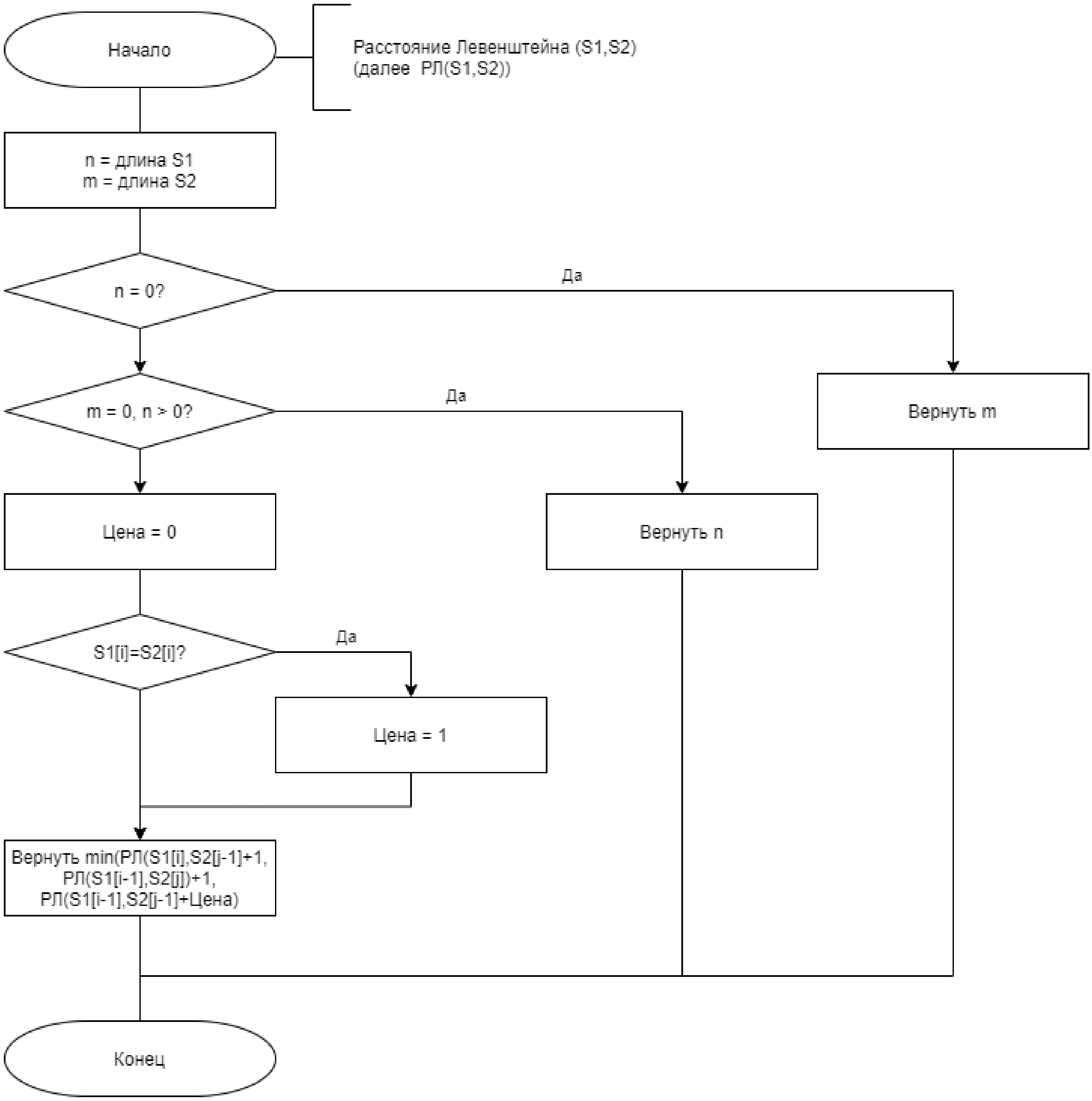


Рисунок 2.1 – Схема рекурсивного алгоритма Левенштейна

На рисунке 2.2 приведена схема рекурсивного алгоритма Левенштейна с кэшированием.

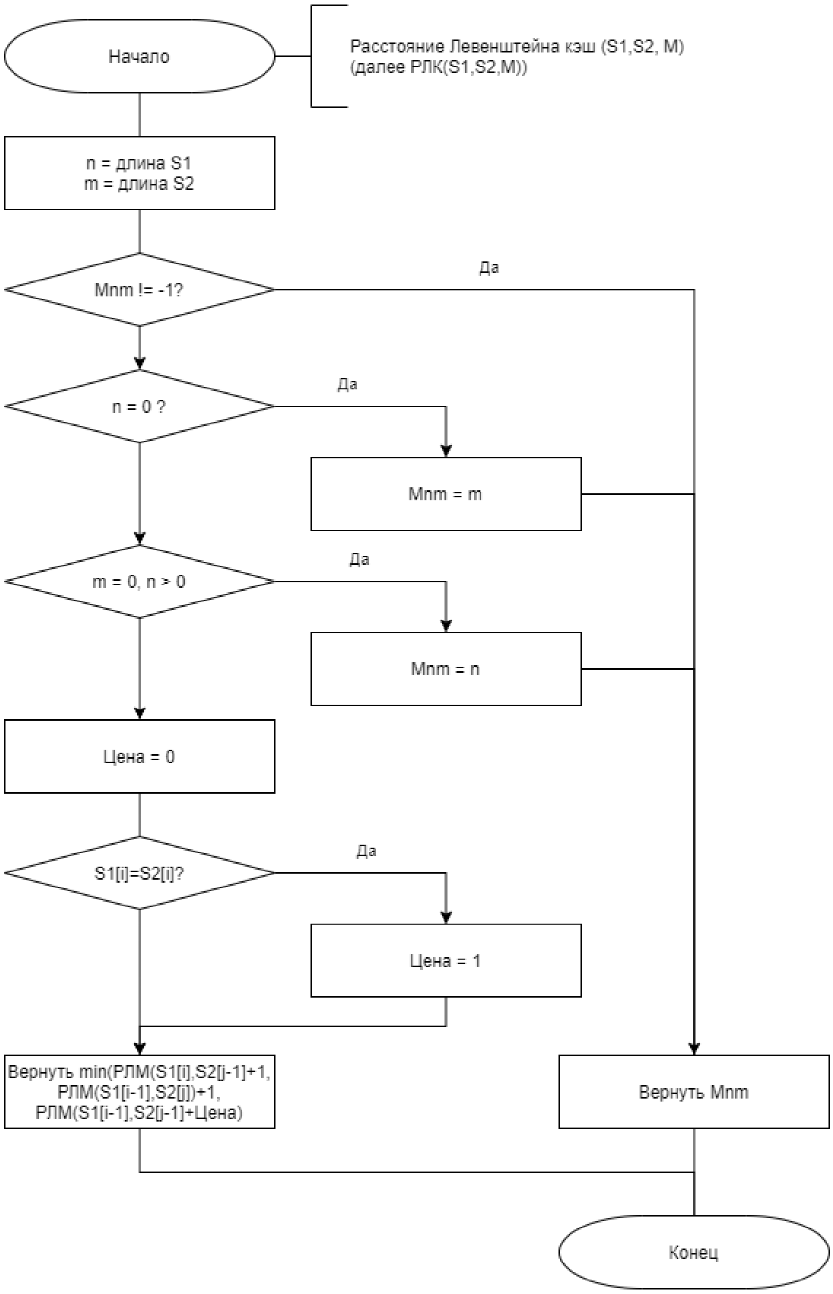


Рисунок 2.2 – Схема рекурсивного алгоритма Левенштейна с кэшированием

2.2 Схема алгоритма Дамерау — Левенштейна

На рисунке 2.3 приведена схема рекурсивного алгоритма Дамерау – Левенштейна.

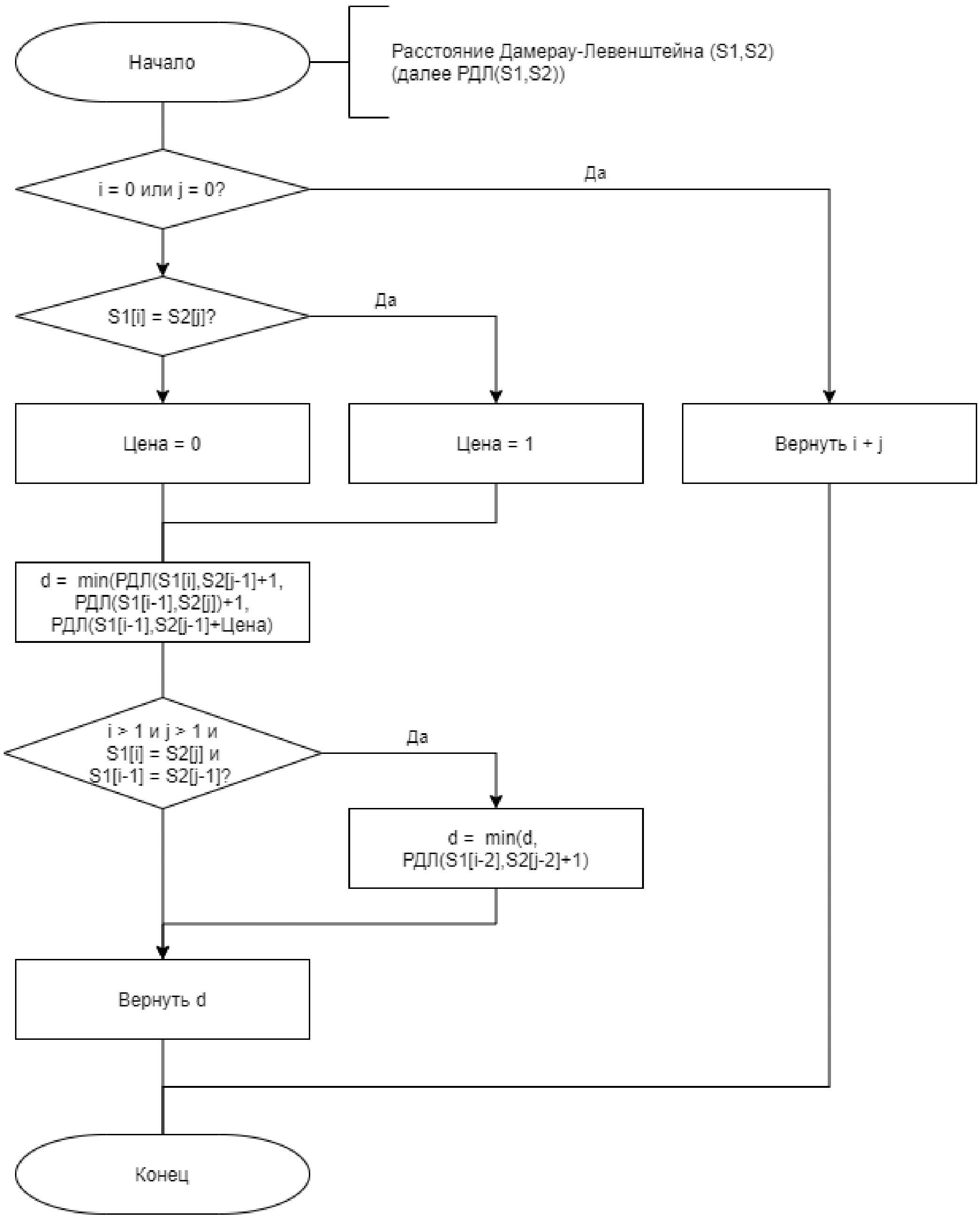


Рисунок 2.3 – Схема рекурсивного алгоритма Дамерау – Левенштейна

На рисунке 2.4 приведена схема матричного алгоритма Дамерау – Левенштейна.

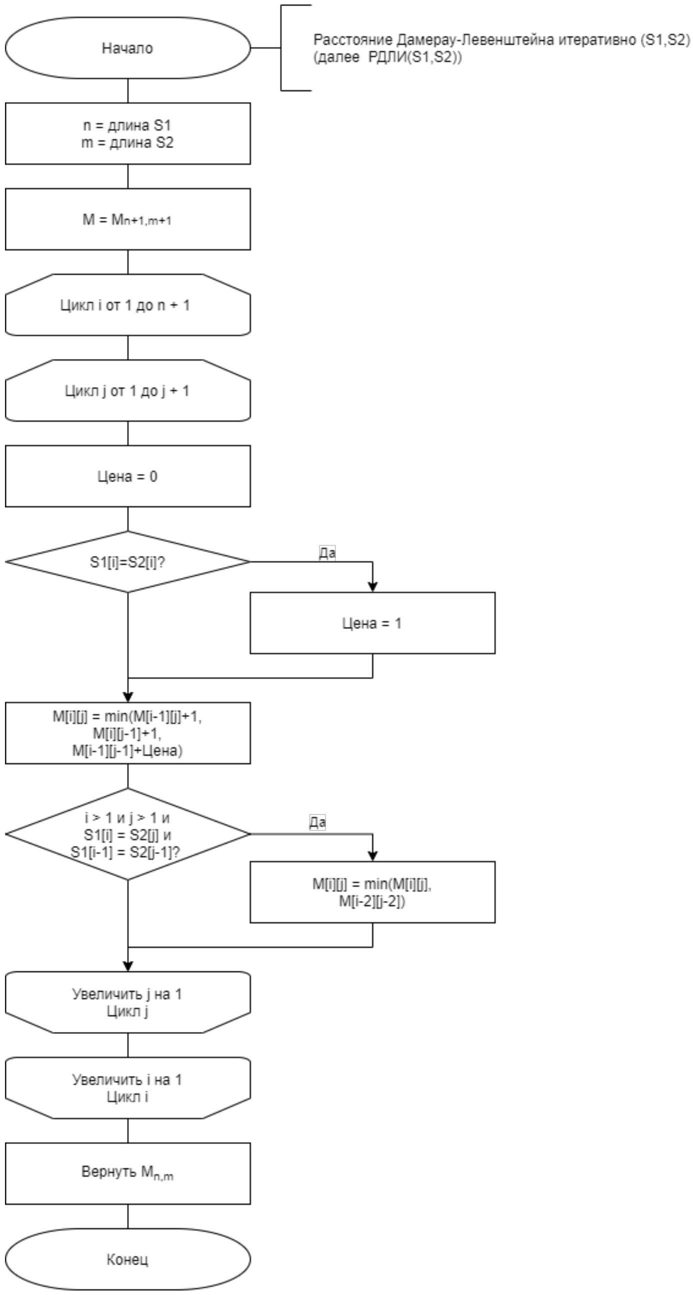


Рисунок 2.4 – Схема итеративного алгоритма Дамерау – Левенштейна

2.3 Вывод

На основе теоретических данных, полученных из аналитического раздела, были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаются две строки в английской или русской раскладе, в том числе содержащие цифры, специальные символы или пустые;
- на выходе — искомое расстояние для всех четырех методов и матрицы расстояний для соотвествующих алгоритмов.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык GO [2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа. Так же данный язык предоставляет средства тестирования разработанного ПО.

3.3 Листинг кода

В листингах 3.1–3.4 приведены реализации алгоритмов Левенштейна и Дамерау — Левенштейна, а также вспомогательные функции.

Листинг 3.1 – Алгоритм Левенштейна

```
func LevRec(str1, str2 string) int {
    s1, s2 := []rune(str1), []rune(str2)

    if len(s1) > len(s2) {
        s1, s2 = s2, s1
    }

    return lev_dist_rec(s1, s2, len(s1), len(s2))
}

func lev_dist_rec(s1, s2 []rune, l1, l2 int) int {
    if l1 == 0 {
        return l2
    }

    if l2 == 0 && l1 > 0 {
        return l1
    }

    check := 1
    if s1[l1-1] == s2[l2-1] {
        check = 0
    }

    return min_value(
        lev_dist_rec(s1, s2, l1, l2-1)+1,
        lev_dist_rec(s1, s2, l1-1, l2)+1,
        lev_dist_rec(s1, s2, l1-1, l2-1)+check)
}
```

Листинг 3.2 – Алгоритм Левенштейна с кэшированием

```
func LevRecCached(str1, str2 string) int {
    s1, s2 := []rune(str1), []rune(str2)

    l1, l2 := len(s1), len(s2)
    if l1 == 0 || l2 == 0 {
        return max_value(l1, l2)
    }

    mtr := levmtr_init(l1, l2, true)

    return dist_rec_cached(s1, s2, len(s1), len(s2), mtr)
}

func dist_rec_cached(s1, s2 []rune, l1, l2 int, mtr [][]int) int {
    if l1 == 0 {
        return l2
    }

    if l2 == 0 {
        return l1
    }

    if mtr[l1-1][l2-1] != -1 {
        return mtr[l1-1][l2-1]
    }

    if s1[l1-1] == s2[l2-1] {
        mtr[l1-1][l2-1] = dist_rec_cached(s1, s2, l1-1, l2-1, mtr)
        return mtr[l1-1][l2-1]
    }

    mtr[l1-1][l2-1] = 1 + min_value(
        dist_rec_cached(s1, s2, l1, l2-1, mtr),
        dist_rec_cached(s1, s2, l1-1, l2, mtr),
```

Листинг 3.3 – Алгоритм Дамерау–Левенштейна

```

    return mtr
}

func DamLevRec(str1, str2 string) int {
    s1, s2 := []rune(str1), []rune(str2)

    if len(s1) > len(s2) {
        s1, s2 = s2, s1
    }

    return damlev_dist_rec(s1, s2, len(s1), len(s2))
}

func damlev_dist_rec(s1, s2 []rune, l1, l2 int) int {
    if l1 == 0 {
        return l2
    }

    if l2 == 0 && l1 > 0 {
        return l1
    }

    check := 0
    if s1[l1-1] != s2[l2-1] {
        check = 1
    }

    if l1 > 1 && l2 > 1 && s1[l1-1] == s2[l2-2] && s1[l1-2] == s2[l2-1] {
        return min_value(
            damlev_dist_rec(s1, s2, l1-1, l2)+1,
            damlev_dist_rec(s1, s2, l1, l2-1)+1,
            damlev_dist_rec(s1, s2, l1-1, l2-1)+check,
            damlev_dist_rec(s1, s2, l1-2, l2-2)+1)
    } else {
        return min_value(
            damlev_dist_rec(s1, s2, l1-1, l2)+1,
            damlev_dist_rec(s1, s2, l1, l2-1)+1,
```

Листинг 3.4 – Алгоритм Дамерау–Левенштейна с матрицей

```

    return dist, mtr
}

func DamLevMtr(str1, str2 string) (int, [][]int) {
    s1, s2 := []rune(str1), []rune(str2)
    l1, l2 := len(s1), len(s2)
    mtr := levmtr_init(l1+1, l2+1, false)
    check := 0
    for i := 1; i < l1+1; i++ {
        for j := 1; j < l2+1; j++ {
            if s1[i-1] == s2[j-1] {
                check = 0
            } else {
                check = 1
            }
            mtr[i][j] = min_value(
                mtr[i-1][j]+1,
                mtr[i][j-1]+1,
                mtr[i-1][j-1]+check)

            if i > 1 && j > 1 && s1[i-1] == s2[j-2] && s1[i-2] == s2[j-1] {
                mtr[i][j] = min_value(mtr[i][j], mtr[i-2][j-2]+1)
            }
        }
    }
    dist := mtr[l1][l2]
    return dist, mtr
}

func levmtr_init(n, m int, is_rec bool) [][]int {
    mtr := make([][]int, n)
    for i := range mtr {
        mtr[i] = make([]int, m)
        mtr[i][0] = i
    }

    for j := 1; j < m; j++ {
        mtr[0][j] = j
    }

    if is_rec {
        for i := range mtr {
            for j := range mtr[i] {
                mtr[i][j] = -1
            }
        }
```

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна. Все тесты пройдены успешно (таблица 3.2).

Таблица 3.1 – Функциональные тесты

Строка 1	**Строка 2**	**Ожидаемый результат**	
		Левенштейн	**Дамерау — Левенштейн**
happy	scrappy	3	3
monster	rodster	2	2
<пусто>	<пусто>	0	0
minute	<пусто>	6	6
suspect	susepct	2	1

Таблица 3.2 – Результат работы программы

Строка 1	**Строка 2**	**Фактический результат**	
		Левенштейн	**Дамерау — Левенштейн**
happy	scrappy	3	3
monster	rodster	2	2
<пусто>	<пусто>	0	0
minute	<пусто>	6	6
suspect	susepct	2	1

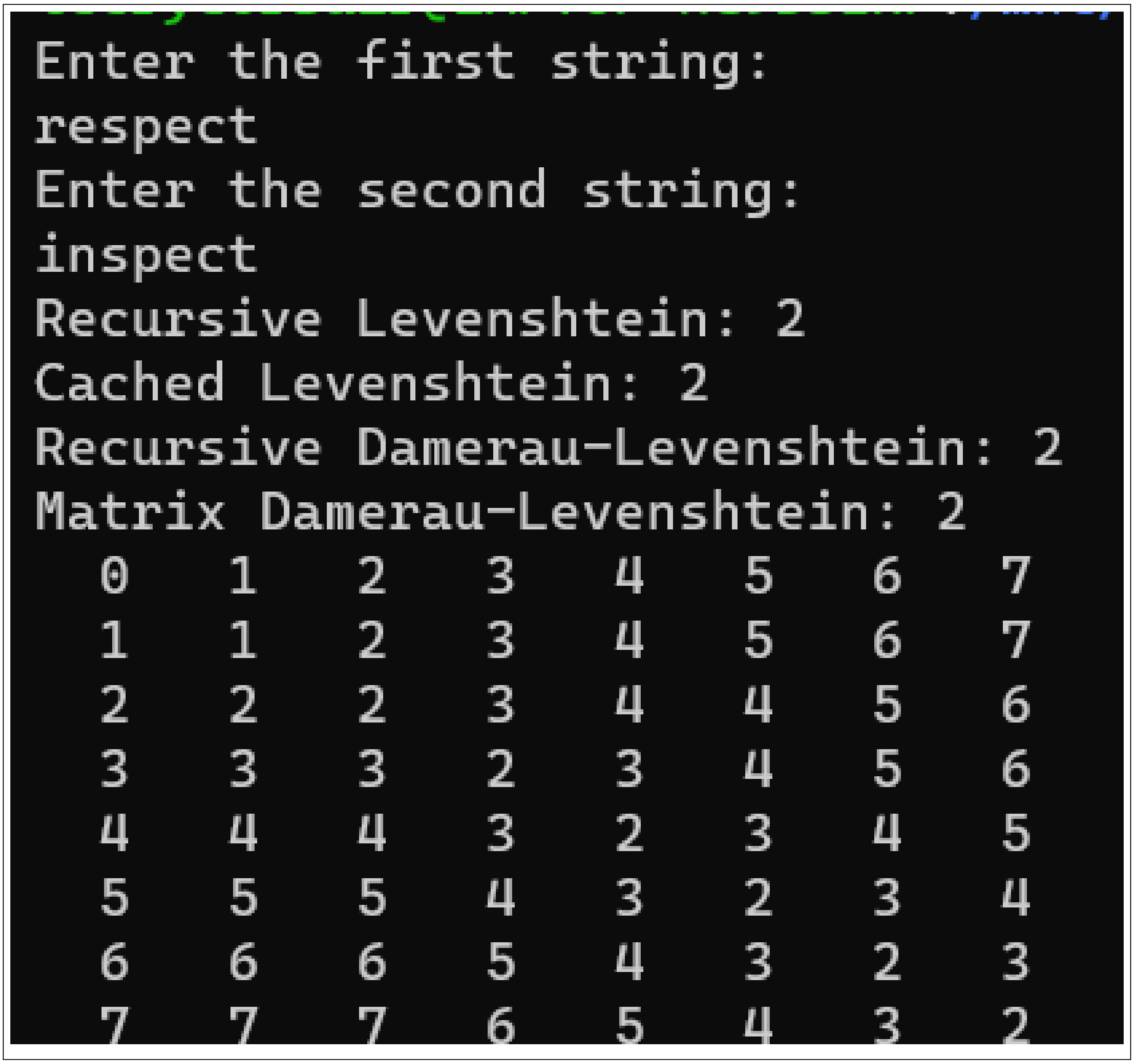
3.4 Вывод

Были разработаны и протестированы спроектированные алгоритмы: вычисления расстояния Левенштейна рекурсивно,рекурсивно с кэшированием, а также вычисления расстояния Дамерау — Левенштейна рекурсивно и с заполнением матрицы.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.



```
Enter the first string:
respect
Enter the second string:
inspect
Recursive Levenshtein: 2
Cached Levenshtein: 2
Recursive Damerau-Levenshtein: 2
Matrix Damerau-Levenshtein: 2
  0   1   2   3   4   5   6   7
  1   1   2   3   4   5   6   7
  2   2   2   3   4   4   5   6
  3   3   3   2   3   4   5   6
  4   4   4   3   2   3   4   5
  5   5   5   4   3   2   3   4
  6   6   6   5   4   3   2   3
  7   7   7   6   5   4   3   2
```

Рисунок 4.1 – Демонстрация работы алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Windows 10 64-bit [3].
- Память: 16 GB.
- Процессор: AMD Ryzen 5 4600H [4] @ 3.00 GHz

Тестирование проводилось на ноутбуке при включённом режиме производительности. Во время тестирования ноутбук был нагружен только системными процессами.

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи написания «бенчмарков» [5], предоставляемых встроенными в Go средствами. Для такого рода тестирования не нужно самостоятельно указывать количество повторов. В библиотеке для тестирования существует константа N, которая динамически варьируется в зависимости от того, был ли получен стабильный результат или нет.

Результаты замеров приведены в таблице 4.1. В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится NaN. На рисунках 4.2 и 4.3 приведены графики зависимостей времени работы алгоритмов от длины строк.

4.4 Использование памяти

Алгоритмы Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, достаточно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов.

Таблица 4.1 – Замер времени для строк, размером от 5 до 100

Длина строк	Время, нс			
	LevRec	**LevCached**	**DamLevRec**	**DamLevMtr**
5	22126	1316	13415	1330
8	2138840	3721	2128415	2719
10	64809090	7320	64095303	3795
12	1992206037	10024	2015092968	5117
20	NaN	23738	NaN	12799
40	NaN	96950	NaN	46122
60	NaN	209082	NaN	100596
80	NaN	368109	NaN	176596
100	NaN	581960	NaN	261581

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти (4.1)

$$(\mathcal{C}(S_1) + \mathcal{C}(S_2)) \cdot (2 \cdot \mathcal{C}(\text{string}) + 3 \cdot \mathcal{C}(\text{int})), \tag{4.1}$$

где $\mathcal{C}$ — оператор вычисления размера, S_1, S_2 — строки, int — целочисленный тип, string — строковый тип.

Использование памяти при итеративной реализации теоретически равно

$$(\mathcal{C}(S_1) + 1) \cdot (\mathcal{C}(S_2) + 1) \cdot \mathcal{C}(\text{int}) + 10 \cdot \mathcal{C}(\text{int}) + 2 \cdot \mathcal{C}(\text{string}). \tag{4.2}$$

Выделение памяти при работе алгоритмов указано на рисунке 4.4.

4.5 Вывод

Рекурсивный алгоритм Левенштейна работает на порядок дольше реализации с кэшированием, время его работы увеличивается в геометрической прогрессии. На словах длиной 12 символов, реализация алгоритма Левенштейна с кэшированием превосходит по времени работы рекурсивную в 200 000 раз. Алгоритм Дамерау — Левенштейна по времени выполнения сопоставим с алгоритмом Левенштейна. В нём добавлены дополнительные

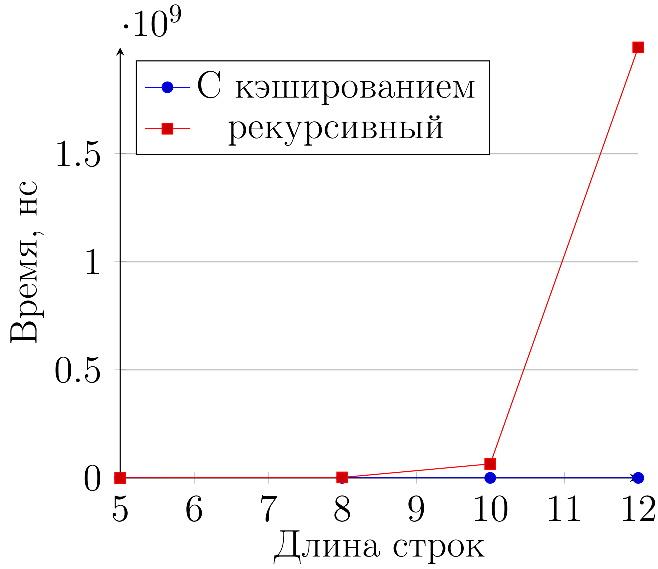


Рисунок 4.2 – Зависимость процессорного времени работы алгоритма вычисления расстояния Левенштейна от длины строк (рекурсивная и с кэшированием реализации)

проверки, и по сути он является алгоритмом другого смыслового уровня. При вычислении расстояния Дамерау–Левенштейна использование итеративного подхода дает выигрышь по времени в 400 000 раз.

Но по расходу памяти итеративные алгоритмы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

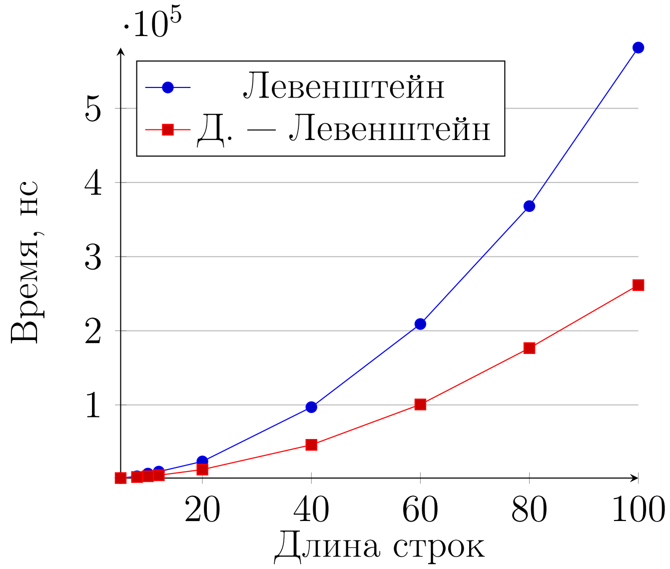


Рисунок 4.3 – Зависимость процессорного времени работы реализаций алгоритмов Левенштейна и Дамерау — Левенштейна(с кэшированием и итеративной реализациями)

```
goos: linux
goarch: amd64
pkg: local/levenshtein
BenchmarkLevRec5-12               235616         22126 ns/op          0 B/op          0 allocs/op
BenchmarkLevRec8-12                 2809       2138840 ns/op         73 B/op          0 allocs/op
BenchmarkLevRec10-12                  91      64809090 ns/op       1571 B/op          0 allocs/op
BenchmarkLevRec12-12                   3    1992206037 ns/op      49152 B/op          6 allocs/op
BenchmarkLevRecCached5-12        4410040          1316 ns/op        368 B/op          6 allocs/op
BenchmarkLevRecCached8-12        1557848          3721 ns/op        704 B/op          9 allocs/op
BenchmarkLevRecCached10-12        814010          7320 ns/op       1040 B/op         11 allocs/op
BenchmarkLevRecCached12-12        746133         10024 ns/op       1440 B/op         13 allocs/op
BenchmarkLevRecCached20-12        234056         23738 ns/op       3680 B/op         21 allocs/op
BenchmarkLevRecCached40-12         59948         96950 ns/op      14147 B/op         43 allocs/op
BenchmarkLevRecCached60-12         28970        209082 ns/op      30825 B/op         63 allocs/op
BenchmarkLevRecCached80-12         16384        368109 ns/op      53904 B/op         83 allocs/op
BenchmarkLevRecCached100-12         9771        581960 ns/op      93143 B/op        103 allocs/op
BenchmarkDamLevMtr5-12           4534916          1330 ns/op        432 B/op          7 allocs/op
BenchmarkDamLevMtr8-12           2205790          2719 ns/op        944 B/op         10 allocs/op
BenchmarkDamLevMtr10-12          1574649          3795 ns/op       1344 B/op         12 allocs/op
BenchmarkDamLevMtr12-12          1000000          5117 ns/op       1776 B/op         14 allocs/op
BenchmarkDamLevMtr20-12           463069         12799 ns/op       4208 B/op         22 allocs/op
BenchmarkDamLevMtr40-12           125614         46122 ns/op      15776 B/op         44 allocs/op
BenchmarkDamLevMtr60-12            60760        100596 ns/op      33248 B/op         64 allocs/op
BenchmarkDamLevMtr80-12            34282        176596 ns/op      59712 B/op         84 allocs/op
BenchmarkDamLevMtr100-12           22687        261581 ns/op      94017 B/op        104 allocs/op
BenchmarkDamLevRec5-12            378536         13415 ns/op          0 B/op          0 allocs/op
BenchmarkDamLevRec8-12              2498       2128415 ns/op        297 B/op          0 allocs/op
BenchmarkDamLevRec10-12               94      64095303 ns/op       1568 B/op          0 allocs/op
BenchmarkDamLevRec12-12                3    2015092968 ns/op      54613 B/op          6 allocs/op
PASS
ok      local/levenshtein    211.819s
```

Рисунок 4.4 – Замеры производительности алгоритмов, выполненные при помощи команды `go test -bench . -benchmem`

Заключение

В ходе выполнения работы были выполнены все поставленные задачи и изучены методы динамического программирования на основе алгоритмов вычисления расстояния Левенштейна.

Экспериментально были установлены различия в производительности различных алгоритмов вычисления расстояния Левенштейна. Рекурсивный алгоритм Левенштейна работает на порядок дольше реализации с кэшированием, время его работы увеличивается в геометрической прогрессии. На словах длиной 12 символов, реализация алгоритма Левенштейна с кэшированием превосходит по времени работы рекурсивную в 200 000 раз. Алгоритм Дамерау — Левенштейна по времени выполнения сопоставим с алгоритмом Левенштейна. В нём добавлены дополнительные проверки, и по сути он является алгоритмом другого смыслового уровня. При вычислении расстояния Дамерау–Левенштейна использование итеративного подхода дает выигрышь по времени в 400 000 раз.

Теоретически было рассчитано использования памяти в каждом из алгоритмов вычисления расстояния Левенштейна. Обычные матричные алгоритмы потребляют намного больше памяти, чем рекурсивные, за счет дополнительного выделения памяти под матрицы и большее количество локальных переменных, однако при длинных строках глубина рекурсии становится слишком большой и рекурсивные алгоритмы без кэширования начинают проигрывать и по памяти.

Список литературы

[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: 01.10.2021).

[3] Explore Windows 10. [Электронный ресурс]. Режим доступа: `https://www.microsoft.com/en-us/windows/` (дата обращения: 02.10.2021).

[4] AMD Processors [Электронный ресурс]. Режим доступа: `https://www.amd.com/en/products/apu/amd-ryzen-5-4600h` (дата обращения: 02.10.2021).

[5] Testing – The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/pkg/testing/` (дата обращения: 08.10.2021).