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 курсу "Анализ алгоритмов"

Тема Редакционные расстояния

Студент Морозов Д.В.

Группа ИУ7-52Б

Оценка (баллы) __________

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Оглавление

Введение

Целью данной лабораторной работы является получение практических навыков динамического программирования на примере реализации алгоритмов Левенштейна и Дамерау-Левенштейна.

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной строки в другую [1].

Расстояние Левенштейна и его обобщения применяются:

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

— для сравнения текстовых файлов утилитой diff и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);

— в биоинформатике [2].

Расстояние Дамерау-Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира Левенштейна) — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна, так как к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов [3].

Задачами данной лабораторной являются:

— изучение и реализация алгоритмов Левенштейна и Дамерау-Левенштейна нахождения редакционного расстояния между строками;

— выполнение теоретической либо экспериментальной оценки затрат алгоритмов по памяти;

— выполнение экспериментальной оценки затрат алгоритмов по времени;

— сравнение алгоритмов по проведённым оценкам.

1 Аналитическая часть

В этом разделе будут представлены описания алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и их практическое применение.

1.1 Матричный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую. При этом каждая операция имеет свою цену (штраф).

Рассмотрим матрицу A размером $(length(s_1)+1)\cdot((length(s_2)+1)$, где $length(S)$ — длина строки S. Пусть значение в ячейке $[i, j]$ матрицы равно расстоянию между префиксом s_1 длины i и префиксом s_2 длины j. У элементов первой строки значение равно индексу столбца, у элементов первого столбца — индексу строки.

Остальные ячейки заполняем в соответствии с формулой (1.1).

$$A[i][j] = min \begin{cases} A[i-1][j] + 1; \\ A[i][j-1] + 1; \\ A[i-1][j-1] + m(s_1[i], s_2[j]). \end{cases} \tag{1.1}$$

Функция m определена как:

$$m(s_1[i], s_2[j]) = \begin{cases} 0, & \text{если } s_1[i-1] = s_2[j-1]; \\ 1, & \text{иначе.} \end{cases} \tag{1.2}$$

В результате расстоянием Левенштейна будет ячейка матрицы с индексами $i = length(s_1)$ и $j = length(s_2)$.

1.2 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна между двумя строками — это минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна.

Рекурсивный алгоритм считает редакционное расстояние для двух строк s_1 и s_2 по рекуррентной формуле (1.3), где i — длина префикса

s_1, j — длина префикса s_2:

$$d(i,j) = \begin{cases} \max(i,j), \text{если } \min(i,j) = 0; \\ \min\{ \\ \qquad d(i,j-1)+1; \\ \qquad d(i-1,j)+1; \\ \qquad d(i-1,j-1)+m(a[i],b[j]); \\ \qquad \begin{cases} d(i-2,j-2)+1, & \text{если } i,j>1, \\ & s_1[i] = s_2[j-1], \\ & s_2[j] = s_1[i-1]; \\ \qquad \infty, & \text{иначе.} \end{cases} \\ \}, \text{иначе.} \end{cases} \tag{1.3}$$

Функция f определена как:

$$m(i,j) = \begin{cases} 0, & \text{если } s_1[i-1] = s_2[j-1]; \\ 1, & \text{иначе.} \end{cases} \tag{1.4}$$

1.3 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием кеша

Рекурсивный алгоритм можно оптимизировать, если записывать найденные промежуточные расстояния в кеш-матрицу. Перед началом рассчёта требуется инициализировать ячейки матрицы значением -1. При рекурсивном вызове требуется проверить, было ли значение вычислено ранее, — проверить, находится ли в соответствующей ячейке матрицы -1. Таким образом, на поиск уже найденных расстояний время не тратится — их значения берутся из матрицы.

1.4 Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

При больших i, j прямая реализация формулы (1.3) может быть неэффективна по времени, так как некоторые значения $D(i, j)$ вычисляются несколько раз. Для оптимизации алгоритма можно хранить промежуточные значения в матрице размером $(length(s_1)+1) \times$ $\times((length(s_2)+1)$, где $length(S)$ — длина строки S. Значение в ячейке $[i, j]$ матрицы A равно расстоянию между префиксом s_1 длины i и префиксом s_2 длины j. У элементов первой строки значение равно индексу столбца, у элементов первого столбца — индексу строки.

Остальные ячейки заполняем в соответствии с формулой (1.5).

$$A[i][j] = min \begin{cases} A[i-1][j] + 1; \\ A[i][j-1] + 1; \\ A[i-1][j-1] + m(s_1[i], s_2[j]); \\ \begin{cases} A[i-2][j-2] + 1, & \text{если } i, j > 1, \\ & s_1[i] = s_2[j-1], \\ & s_2[j] = s_1[i-1]; \\ \infty, & \text{иначе.} \end{cases} \end{cases} \tag{1.5}$$

Функция m определена как

$$m(s_1[i], s_2[j]) = \begin{cases} 0, & \text{если } s_1[i-1] = s_2[j-1]; \\ 1, & \text{иначе.} \end{cases} \tag{1.6}$$

В результате расстоянием Дамерау-Левенштейна будет ячейка матрицы с индексами $i = length(s_1)$ и $j = length(s_2)$.

2 Конструкторская часть

В этом разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

2.1 Матричный алгоритм поиска расстояния Левенштейна

На рисунке 2.1 приведена схема матричного алгоритма нахождения расстояния Левенштейна.

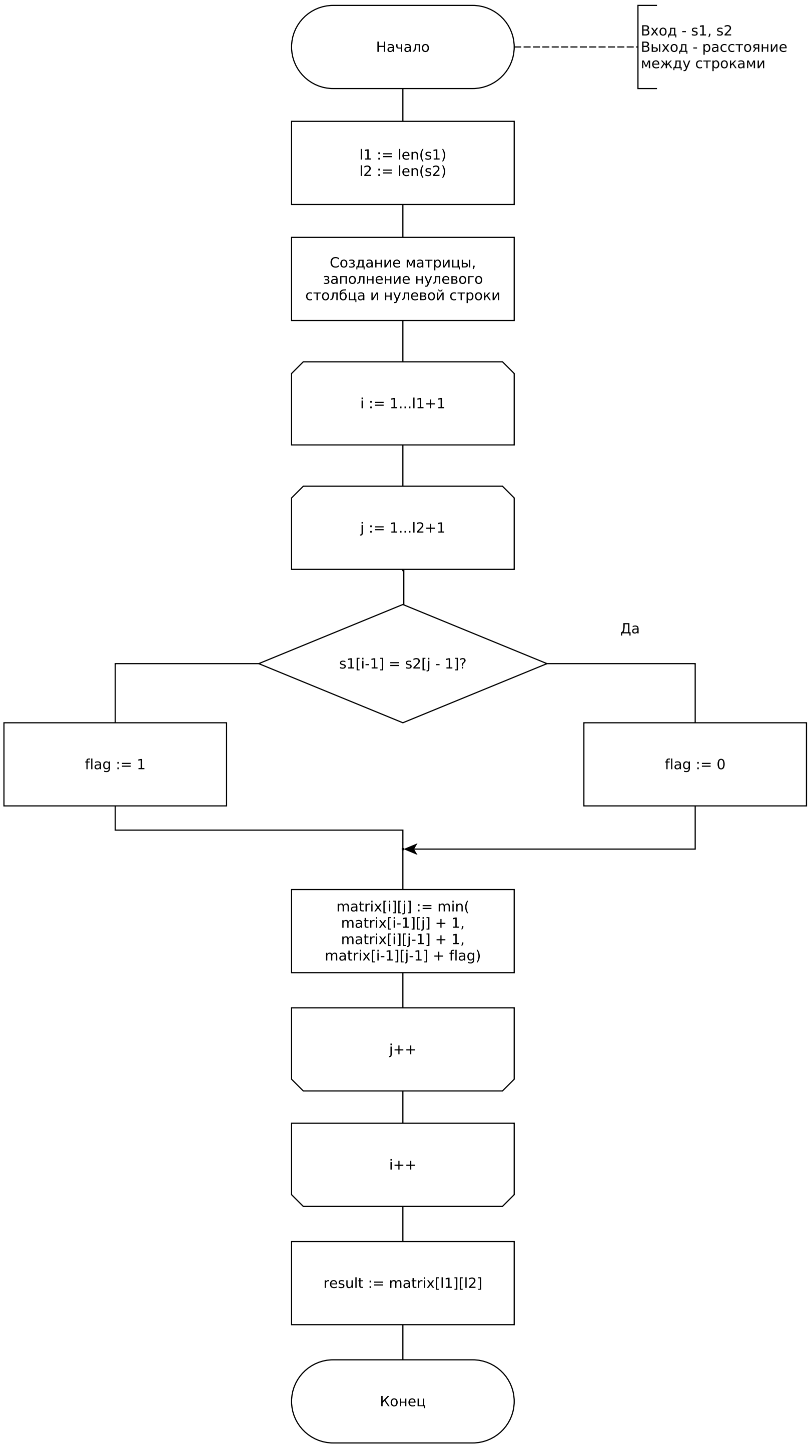

Рисунок 2.1 – Схема матричного алгоритма нахождения расстояния Левенштейна

2.2 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

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

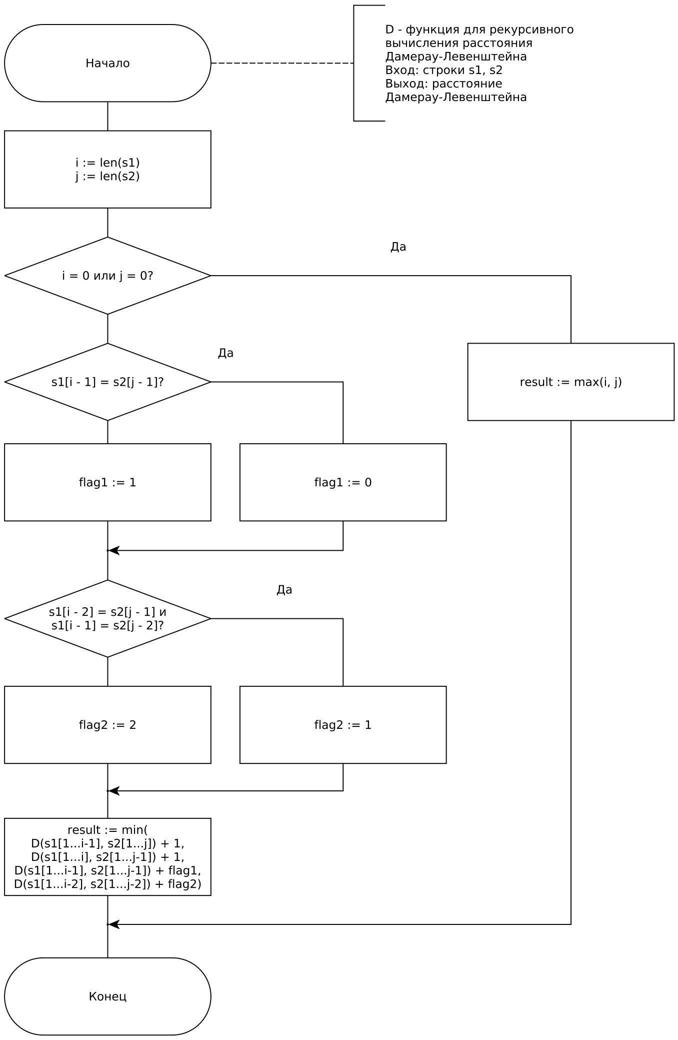

Рисунок 2.2 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

2.3 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием кеша

На рис. 2.3 приведена схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с использованием кеша — матрицы.

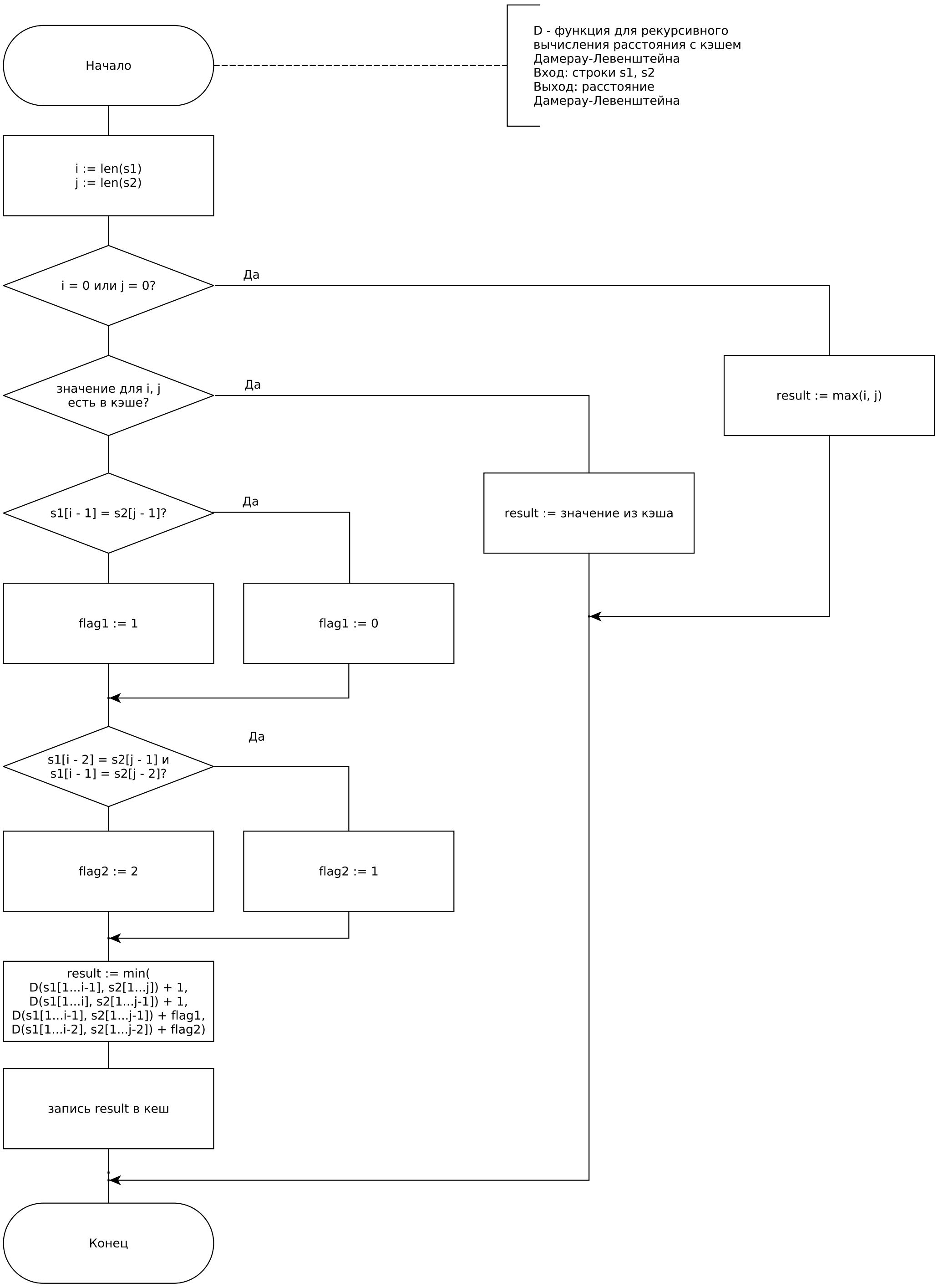

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с использованием кеша (матрицы)

2.4 Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

На рисунке 2.4 приведена схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна.

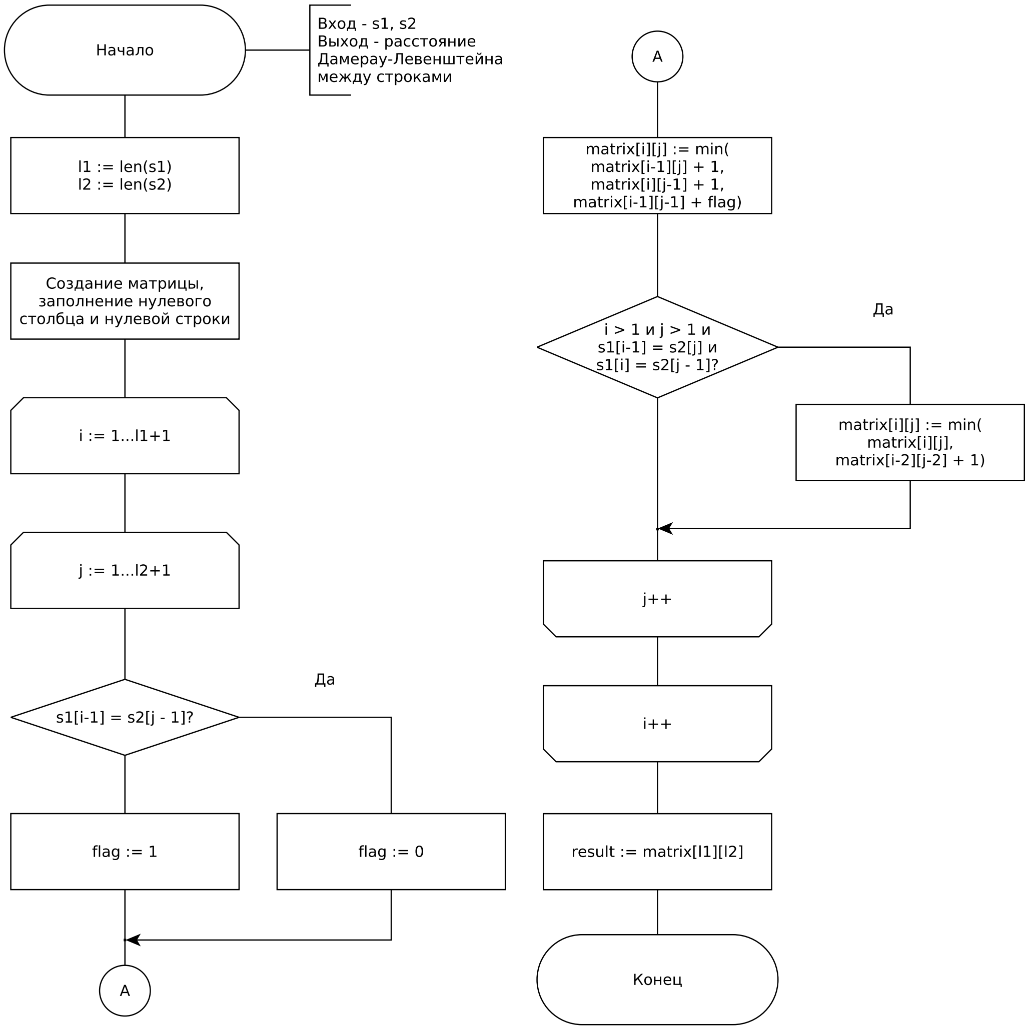

Рисунок 2.4 – Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3 Технологическая часть

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

3.1 Требования к ПО

Используемое ПО должно предоставлять возможность измерения процессорного времени.

3.2 Средства реализации

Для реализации данной лабораторной работы был выбран язык программирования Golang [4] и среда разработки Goland, которая позволяет замерять процессорное время с помощью пакета `CGo` [5].

3.3 Листинг кода

В листингах 3.1–3.6 приведены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна с заполнением матрицы (начало)

```
func LevenshteinMatrix(str1, str2 string) int {
    n := len(str1)
    m := len(str2)
    if n == 0 {
        return m
    } else if m == 0 {
        return n
    }

    b1 := make([]int, n+1)
    b2 := make([]int, n+1)

    for i := 0; i < n+1; i++ {
        b2[i] = i
    }
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна с заполнением матрицы (окончание)

```
    for i := 1; i < m+1; i++ {
        swap(&b1, &b2)

        b2[0] = i
        for j := 1; j < n+1; j++ {
            flag := 1
            if str1[j-1] == str2[i-1] {
                flag = 0
            }

            res := min(b2[j-1]+1, b1[j]+1, b1[j-1]+flag)

            b2[j] = res
        }
    }

    return b2[n]
}
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна без использованиея рекурсии (начало)

```
func DamerauLevenshteinMatrix(str1, str2 string) int {
    n := len(str1)
    m := len(str2)

    if n == 0 {
        return m
    } else if m == 0 {
        return n
    }
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна без использованиея рекурсии (окончание)

```
    b1 := make([]int, n+1)
    b2 := make([]int, n+1)
    b3 := make([]int, n+1)

    for i := 0; i < n+1; i++ {
        b3[i] = i
    }

    for i := 1; i < m+1; i++ {
        swapLeft(&b1, &b2, &b3)

        b3[0] = i
        for j := 1; j < n+1; j++ {
            flag := 1
            if str1[j-1] == str2[i-1] {
                flag = 0
            }

            res := min(b3[j-1]+1, b2[j]+1, b2[j-1]+flag)

            if i > 1 && j > 1 && str1[j-1] == str2[i-2] &&
               str1[j-2] == str2[i-1] {
                res = min(res, b1[j-2] + 1)
            }

            b3[j] = res
        }
    }

    return b3[n]
}
```

Листинг 3.5 – Функция нахождения расстояния Дамерау-Левенштейна с использованием рекурсии.

```
func DamerauLevenshteinRec(str1, str2 string) int {
    n := len(str1)
    m := len(str2)

    if n == 0 {
        return m
    } else if m == 0 {
        return n
    }

    flag := 1
    if str1[n-1] == str2[m-1] {
        flag = 0
    }

    res := min(DamerauLevenshteinRec(str1[:n-1], str2)+1,
    DamerauLevenshteinRec(str1, str2[:m-1])+1,
    DamerauLevenshteinRec(str1[:n-1], str2[:m-1])+flag)

    if n >= 2 && m >= 2 && str1[n-1] == str2[m-2] &&
        str1[n-2] == str2[m-1] {
        cur := DamerauLevenshteinRec(str1[:n-2], str2[:m-2])
            + 1
        if cur < res {
            res = cur
        }
    }

    return res
}
```

Листинг 3.6 – Функция нахождения расстояния Дамерау-Левенштейна с использованием рекурсии с кешем (начало)

```
func damerauLevenshteinRecCache(str1, str2 string, cache
    [][]int) int {
     n := len(str1)
    m := len(str2)

     if n == 0 {
          return m
     } else if m == 0 {
          return n
     }

     if cache[n-1][m-1] != -1 {
          return cache[n-1][m-1]
     }

     res := damerauLevenshteinRecCache(str1[:n-1], str2,
        cache) + 1

     cur := damerauLevenshteinRecCache(str1, str2[:m-1],
        cache) + 1
     if cur < res {
          res = cur
     }

     flag := 1
     if str1[n-1] == str2[m-1] {
          flag = 0
     }

     cur = damerauLevenshteinRecCache(str1[:n-1], str2[:m-1],
        cache) + flag
```

Листинг 3.7 – Функция нахождения расстояния Дамерау-Левенштейна с использованием рекурсии с кешем (окончание)

```
    if cur < res {
        res = cur
    }

    if n >= 2 && m >= 2 && str1[n-1] == str2[m-2] &&
       str1[n-2] == str2[m-1] {
        cur = damerauLevenshteinRecCache(str1[:n-2],
            str2[:m-2], cache) + 1
        if cur < res {
            res = cur
        }
    }

    cache[n-1][m-1] = res
    return res
}
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна (в таблице столбец подписан "Левенштейн") и Дамерау-Левенштейна (в таблице — "Дамерау-Л."). Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

	Входные данные		Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1	cat	cute	2	2
2	cute	cat	2	2
3	dog	dog	0	0
4	toook	t	4	4
5	пустая строка	пустая строка	0	0
8	пустая строка	33	2	2
9	1234	пустая строка	4	4
10	sample	samlpee	2	2
11	abcde	abced	2	1
12	abcde	based	4	2

4 Исследовательская часть

В данном разделе произведено сравнение алгоритмов.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени:

— операционная система — Ubuntu 22.04.1 Linux x86_64;

— оперативная память — 8 ГБ;

— процессор — AMD Ryzen 5 3550H [6].

Замеры проводились на ноутбуке, включенном в сеть электропитания. Во время замеров ноутбук не был нагружен сторонними приложениями.

4.2 Время выполнения алгоритмов

На рисунке 4.1 представлен график, иллюстрирующий зависимость времени работы от длины строк для матричного алгоритма поиска расстояния Левенштейна и матричного алгоритма поиска расстояния Дамерау-Левенштейна.

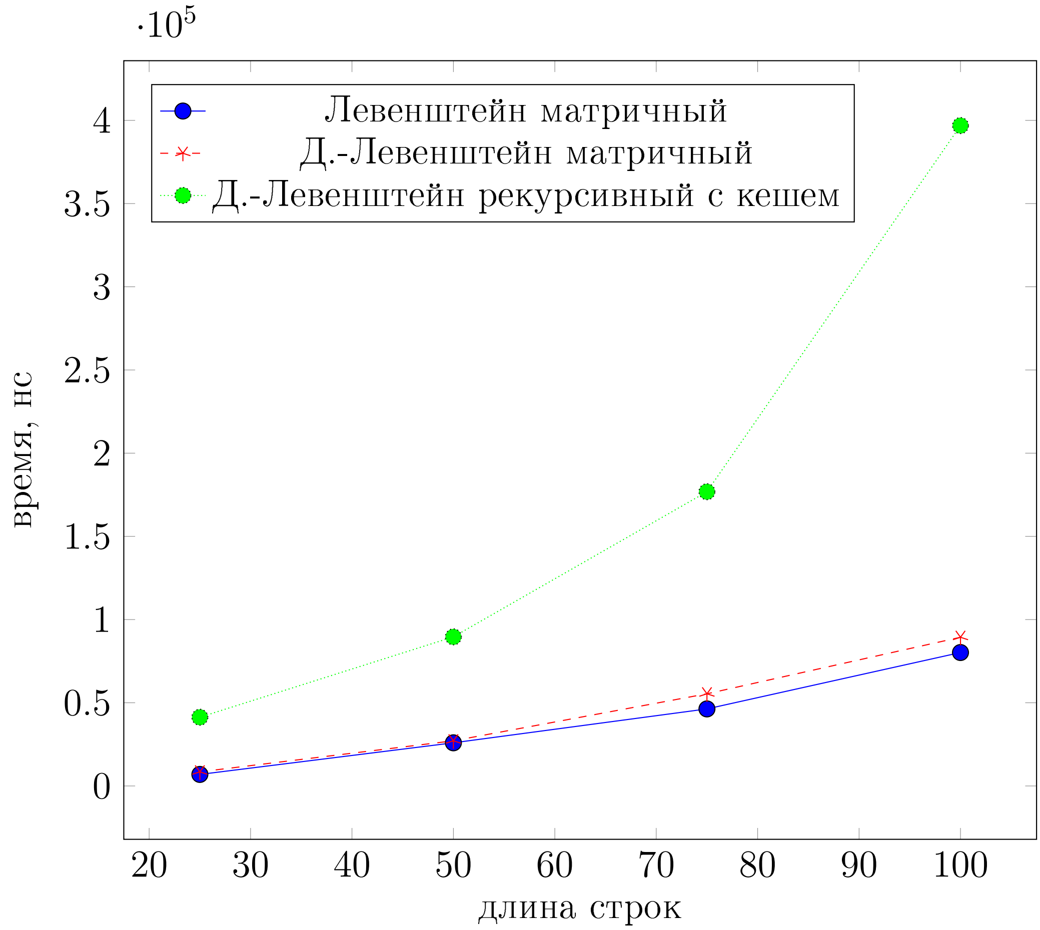

Рисунок 4.1 – Сравнение матричного алгоритма поиска расстояния Левенштейна и матричного алгоритма поиска расстояния Дамерау-Левенштейна

На рисунке 4.2 представлен график, иллюстрирующий зависимость времени работы от длины строк для рекурсивных алгоритмов поиска расстояния Дамерау-Левенштейна с использованием кеша и без.

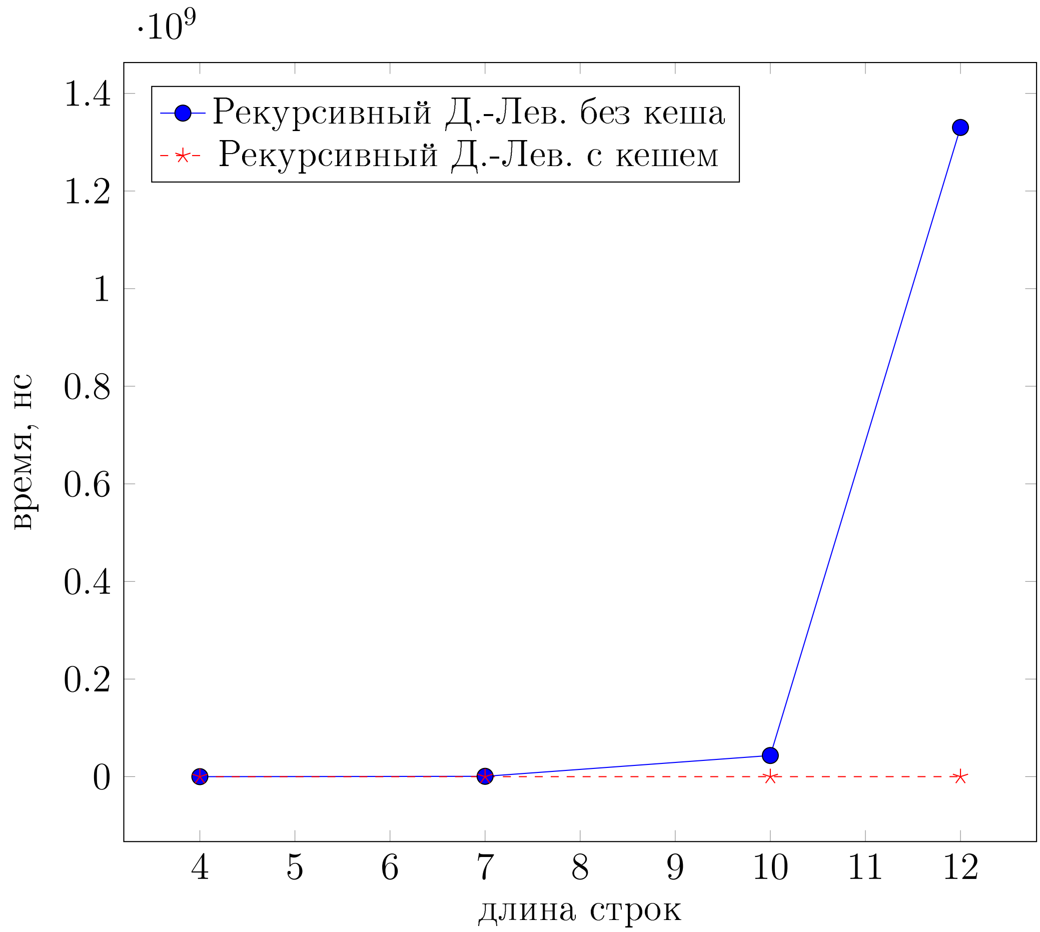

Рисунок 4.2 – Сравнение рекурсивных алгоритмов поиска расстояния Дамерау-Левенштейна с использованием кеша и без

4.3 Использование памяти

Пусть длина строки S_1 — n, длина строки S_2 — m. Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящий строк.

Обозначим: char — тип, используемый для хранения символа строки, int — тип, используемый для хранения чисел.

Рассчитаем затраты по памяти для матричных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна:

— строки S_1, S_2 — $(m + n) \cdot sizeof(char)$;

— матрица — $(m + 1) \cdot (n + 1) \cdot sizeof(int)$;

— длины строк — $2 \cdot sizeof(int)$;

— вспомогательные переменные — $3 \cdot sizeof(int)$;

— итого — $(m+n) \cdot sizeof(char) + (m+1) \cdot (n+1) \cdot sizeof(int) + 5 \cdot sizeof(int)$.

Рассчитаем затраты по памяти для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна(для каждого вызова):

— строки S_1, S_2 — $(m + n) \cdot sizeof(char)$;

— длины строк — $2 \cdot sizeof(int)$;

— вспомогательные переменные — $3 \cdot sizeof(int)$;

— адрес возврата — 8 байт;

— итого — $(m + n) \cdot sizeof(char) + 5 \cdot sizeof(int) + 8$ байт.

Высота дерева рекурсивных вызовов $max(m, n) + 1$. Тогда максимальная глубина стека равна

$$\begin{gathered} M_{rec} = (min(m, n) + 1) \cdot ((m + n) \cdot sizeof(char) + \\ +5 \cdot sizeof(int) + 8). \end{gathered} \tag{4.1}$$

Рассчитаем затраты по памяти для рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с использованием кеша (для каждого вызова):

— строки S_1, S_2 — $(m + n) \cdot sizeof(char)$;

— длины строк — $2 \cdot sizeof(int)$;

— вспомогательные переменные — $3 \cdot sizeof(int)$;

— ссылка на матрицу — 8 байт;

— адрес возврата — 8 байт;

— итого — $(m + n) \cdot sizeof(char) + 5 \cdot sizeof(int) + 16$ байт.

Память для хранения матрицы(для всех вызовов общая)

$$M_{matr} = (m + 1) \cdot (n + 1) \cdot sizeof(int). \tag{4.2}$$

Максимальная глубина стека равна

$$\begin{gathered} M_{cash} = (min(m, n) + 1) \cdot ((m + n) \cdot sizeof(char) + \\ +5 \cdot sizeof(int) + 16) + (m + 1) \cdot (n + 1) \cdot sizeof(int). \end{gathered} \tag{4.3}$$

Вывод

Алгоритм нахождения расстояния Дамерау-Левенштейна по времени выполнения незначительно отличается от алгоритма нахождения расстояния Левенштейна (для слов длиной 100 символов 89 мкс против 80 мкс).

По расходу памяти итеративный алгоритм проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Рекурсивный алгоритм с заполнением матрицы превосходит по времени работы простой рекурсивный (для слов длиной 10 символов 16 мкс против 43353 мкс) и сравним с матричным алгоритмом.

Заключение

Цель достигнута. В ходе выполнения лабораторной работы были решены следующие задачи:

— изучены и реализованы алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;

— выполнена теоретическая оценка затрат алгоритмов по памяти;

— выполнена экспериментальная оценка затрат алгоритмов по времени;

— проведено сравнение алгоритмов по проведённым оценкам.

Алгоритм нахождения расстояния Дамерау-Левенштейна по производительности схож с алгоритмом нахождения расстояния Левенштейна (для слов длиной 100 быстрее на 9%).

Рекурсивный алгоритм с заполнением матрицы эффективнее по времени работы, чем простой рекурсивный (для слов длиной 10 в 2700 раз быстрее) и незначительно отличается от матричной реализации (для слов длиной 100 в 4 раз медленнее). Однако по расходу памяти рекурсивные алгоритмы эффективнее матричных, так как максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Список использованных источников

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

2. А.С.Гуменюк Н.Н.Поздниченко И.Н.Родионов С.Н.Шпынов. О средствах формального анализа строя нуклеотидных цепей // Математическая биология и биоинформатика. Омский государственный технический университет, 2013. Т. 8. С. 373–397.

3. Черненький В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным // Вестник МГТУ им. Н.Э. Баумана. Сер. "Приборостроение". – М.: Издательство МГТУ им. Н.Э. Баумана., 2012. Т. 163. С. 30–34.

4. The Go Programming Language [Электронный ресурс]. Режим доступа: `https://golang.org/` (дата обращения: 14.09.2022).

5. C? Go? Cgo! – The Go Programming Language [Электронный ресурс]. Режим доступа: `https://go.dev/blog/cgo` (дата обращения: 14.09.2022).

6. AMD Ryzen™ 5 3550H [Электронный ресурс]. Режим доступа: `https://www.amd.com/en/products/apu/amd-ryzen-5-3550h` (дата обращения: 14.09.2022).