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 №44
по дисциплине «Анал алгоритмов»

Тема: Редакционные расстояния-хуяния
Студент: Тупыш А. В.
Группа: ИУ7-52Б
Оценка (баллы): _____
Преподаватели: Волкова Л. Л., Душнила Ю. В.

Самара — 2022 г.

Оглавление

Введение

Расстояние Левенштейна между двумя строками — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения строки в другую [1].

Рассматриваются следующие редакционные операции:

— вставка одного символа;

— удаление одного символа;

— замена символа на другой.

Расстояние Дамерау-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.

Редакционные расстояния находят свое применение в:

— компьютерной лингвистике;

— биоинформатике (сравнение генов).

Цель работы

Получение навыков кодирования программного продукта, тестирования и проведения замерного эксперимента работы программы на различных данных. Все это на примере решения задачи нахождения редакционного расстояния.

Задачи работы

1) изучение алгоритмов вычисления редакционного расстояния:

   — Левенштейна;

   — Дамерау-Левенштейна;

   — рекурсивного Дамерау-Левенштейна;

   — рекурсивного Дамерау-Левенштейна с кешем;

2) кодирование данных алгоритмов;

3) проведение замерного эксперимента для данных алгоритмов, с измерением времени работы и использования памяти;

4) проведение сравнительного анализа алгоритмов на основе полученных данных.

1. Аналитическая часть

В данном разделе рассмотрены теоретические выкладки по алгоритмам поиска редакционного расстояния, а также общие сведение о данной задаче.

Так как за редакционное расстояние принято минимальное количество редакционных операций над строкой, с помощью которых она может быть преобразована в другую строку. Для вычисление данной характеристики для каждой операции вводится условная цена [1]:

— $w(a, a) = 0$ — цена замены символа на него же;

— $w(a, b) = 1, a \neq b$ — цена замены одного символа на другой;

— $w(\epsilon, a) = 0$ — цена вставки символа a;

— $w(a, \epsilon) = 0$ — цена удаления символа a.

Для описания алгоритма вычисления расстояния Левенштейна применяется рекуррентное соотношение

$$D(s_1[1..i], s_2[1..j]) = \begin{cases} 0, & \text{если } i = 0, j = 0, \\ i, & \text{если } i > 0, j = 0, \\ j, & \text{если } i = 0, j > 0, \\ D_{min}, & \text{иначе.} \end{cases} \tag{1.1}$$

При этом

$$D_{min} = \min \begin{cases} D(s_1[1..i], s_2[1..j-1]) + 1, \\ D(s_1[1..i-1], s_2[1..j]) + 1, \\ D(s_1[1..i-1], s_2[1..j-1]) + r, \end{cases} \tag{1.2}$$

$$r = \begin{cases} 0, & \text{если } s_1[i] = s_2[j], \\ 1, & \text{иначе.} \end{cases} \tag{1.3}$$

Для алгоритма вычисления расстояния Дамерау-Левенштейна добавляется ещё один элемент:

$$D(s_1[1..i], s_2[1..j]) = \begin{cases} 0, & \text{если } i = 0, j = 0, \\ i, & \text{если } i > 0, j = 0, \\ j, & \text{если } i = 0, j > 0, \\ D_{min}, & \text{иначе.} \end{cases} \tag{1.4}$$

При этом

$$D_{min} = \min \begin{cases} D(s_1[1..i], s_2[1..j-1]) + 1, \\ D(s_1[1..i-1], s_2[1..j]) + 1, \\ D(s_1[1..i-1], s_2[1..j-1]) + r, \\ D(s_1[1..i-2], s_2[1..j-2]) + k, \qquad \text{если } i > 1, j > 1, \end{cases} \tag{1.5}$$

$$r = \begin{cases} 0, & \text{если } s_1[i] = s_2[j], \\ 1, & \text{иначе,} \end{cases} \tag{1.6}$$

$$k = \begin{cases} 0, & \text{если } s_1[i] = s_2[j] \text{ и } s_1[i-1] = s_2[j-1], \\ 1, & \text{если } s_1[i] = s_2[j-1] \text{ и } s_1[i-1] = s_2[j], \\ \infty, & \text{иначе.} \end{cases} \tag{1.7}$$

Могут быть использованы два алгоритма: матричный и рекурсивный.

Под матричным подходом понимается реализация, при которой сначала вычисляются значения для элементарных случаев, а на основе них вычисляется значение для исходных строк.

Под рекурсивным подходом понимается реализация, при которой сначала вычисляется редакционное расстояние от исходных строк, и все необходимые для этого данные вычисляются вложенно. Рекурсивный метод представляет собой перенос математического представления функции в код.

1.1. Матричный алгоритм нахождения расстояния Левенштейна

В матричном алгоритме нахождения расстояния Левенштейна используется формула (1.1). Для того, чтобы не проводить повторные вычисления, промежуточные значения вычисления помещаются в матрицу M. Таким образом, значение в ячейке $M[i][j]$ соответствует расстоянию Левенштейна между строками $s_1[1..i]$ и $s_2[1..j]$.

1.2. Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

В матричном алгоритме нахождения расстояния Дамерау-Левенштейна используется формула (1.4). Для того, чтобы не проводить повторные вычисления, промежуточные значения вычисления помещаются в матрицу M. Таким образом, значение в ячейке $M[i][j]$ соответствует расстоянию Дамерау-Левенштейна между строками $s_1[1..i]$ и $s_2[1..j]$.

1.3. Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

В рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна используется формула (1.4). При этом, в отличие от метода, описанного в 1.2., матрица промежуточных значения не используется, и значения расстояний Дамерау-Левенштейна вычисляются с использованием рекурсивных вызовов.

Данный метод является неэффективным с точки зрения времени выполнения и потребления памяти, так как:

— на рекурсивный вызов затрачивается дополнительное время [2];

— на рекурсивный вызов выделяется стек под вызванную функцию [2];

— производится много повторных вычислений [1].

1.4. Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с кешированием

Данный метод является оптимизацией алгоритма 1.3., в котором используется матрица промежуточных вычислений, аналогичная 1.2.. Использование матрицы позволяет повысить эффективность работы алгоритма с точки зрения времени выполнения и потребления памяти путем исключения повторных вызовов.

Сначала все ячейки матрицы заполняются значением ∞. Затем, по мере вычисления промежуточных значений, заполняются соответствующие ячейки матрицы. И если при шаге рекурсии значение в ячейке матрицы не равно ∞, то вместо повторного вычисления текущего значения, результат берется из матрицы.

2. Конструкторская часть

В данном разделе представлены схемы алгоритмов поиска редакционного расстояния.

2.1. Разработка матричного алгоритма нахождения расстояния Левенштейна

На рисунке 2.1 представлена схема матричного алгоритма нахождения расстояния Левенштейна.

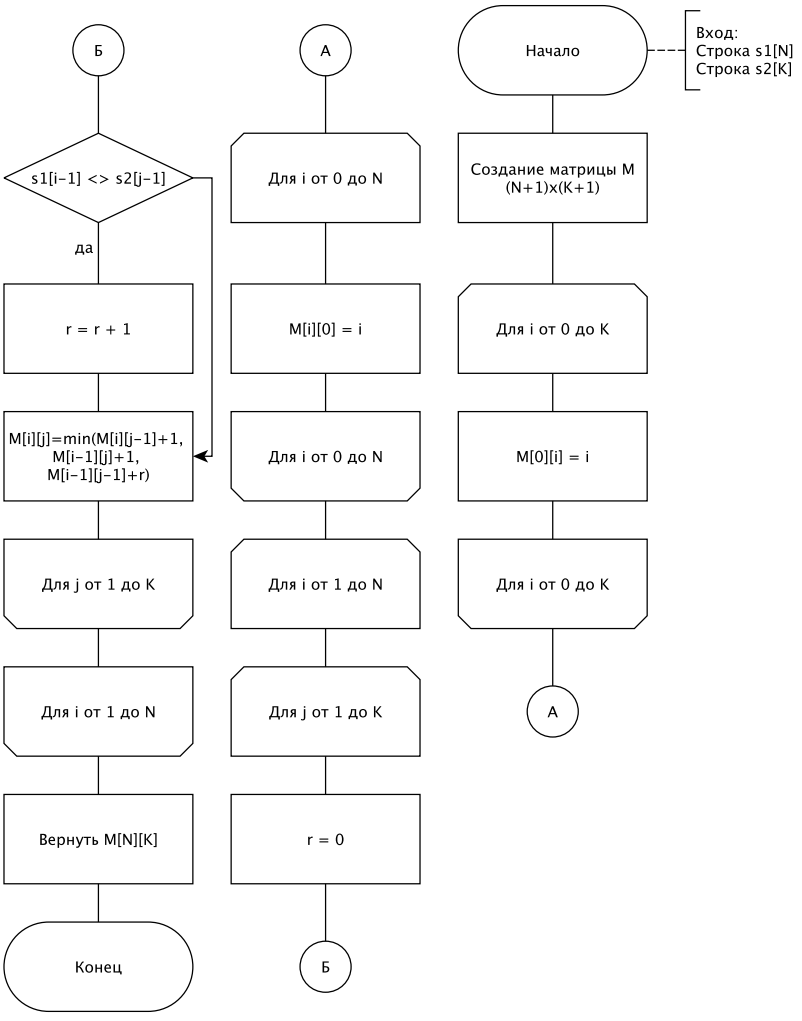

Рисунок 2.1 — Схема матричного алгоритма нахождения расстояния Левенштейна

2.2. Разработка матричного алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунке 2.2 представлена схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна.

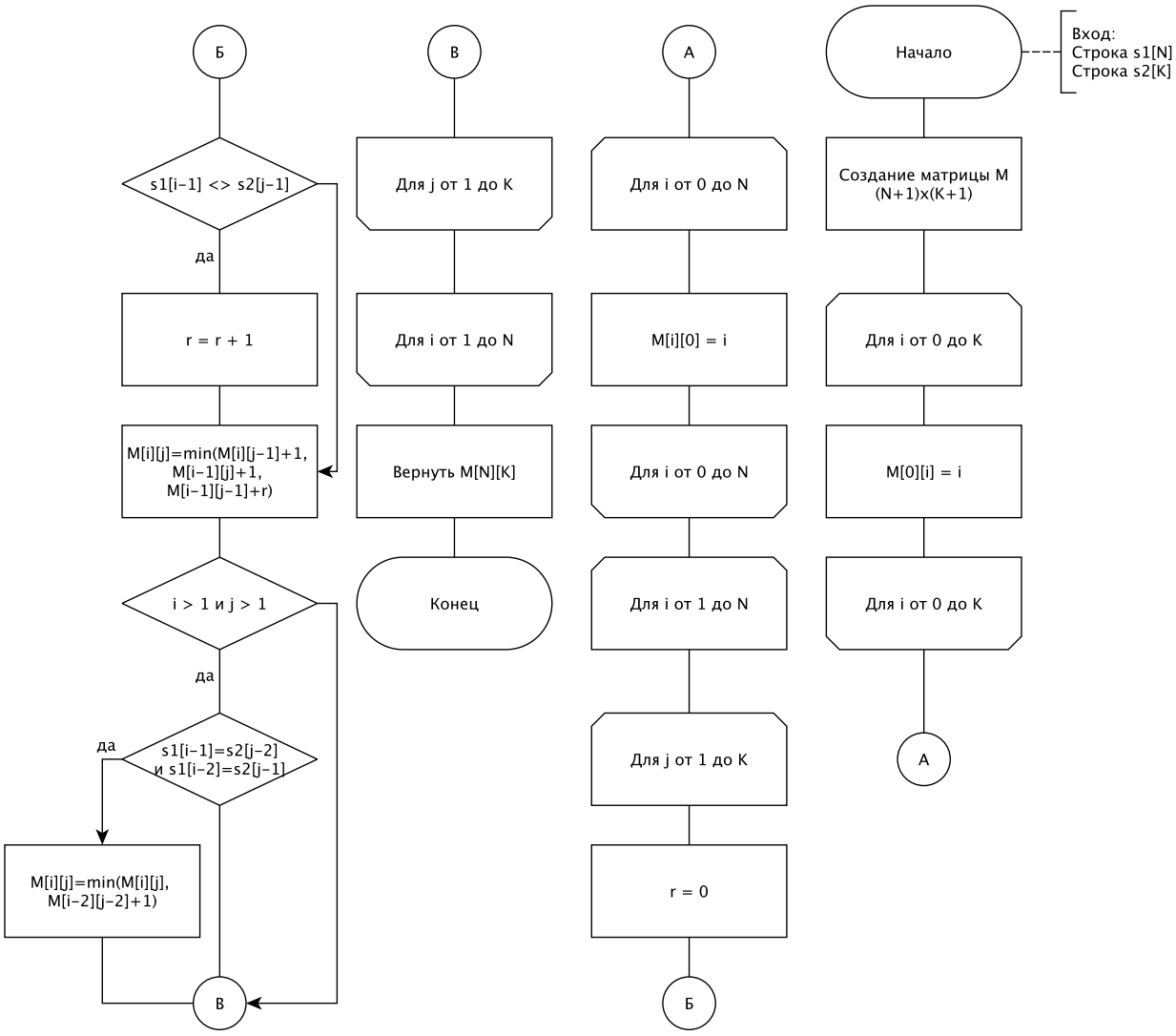

Рисунок 2.2 — Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

2.3. Разработка рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

На рисунке 2.3 представлена схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

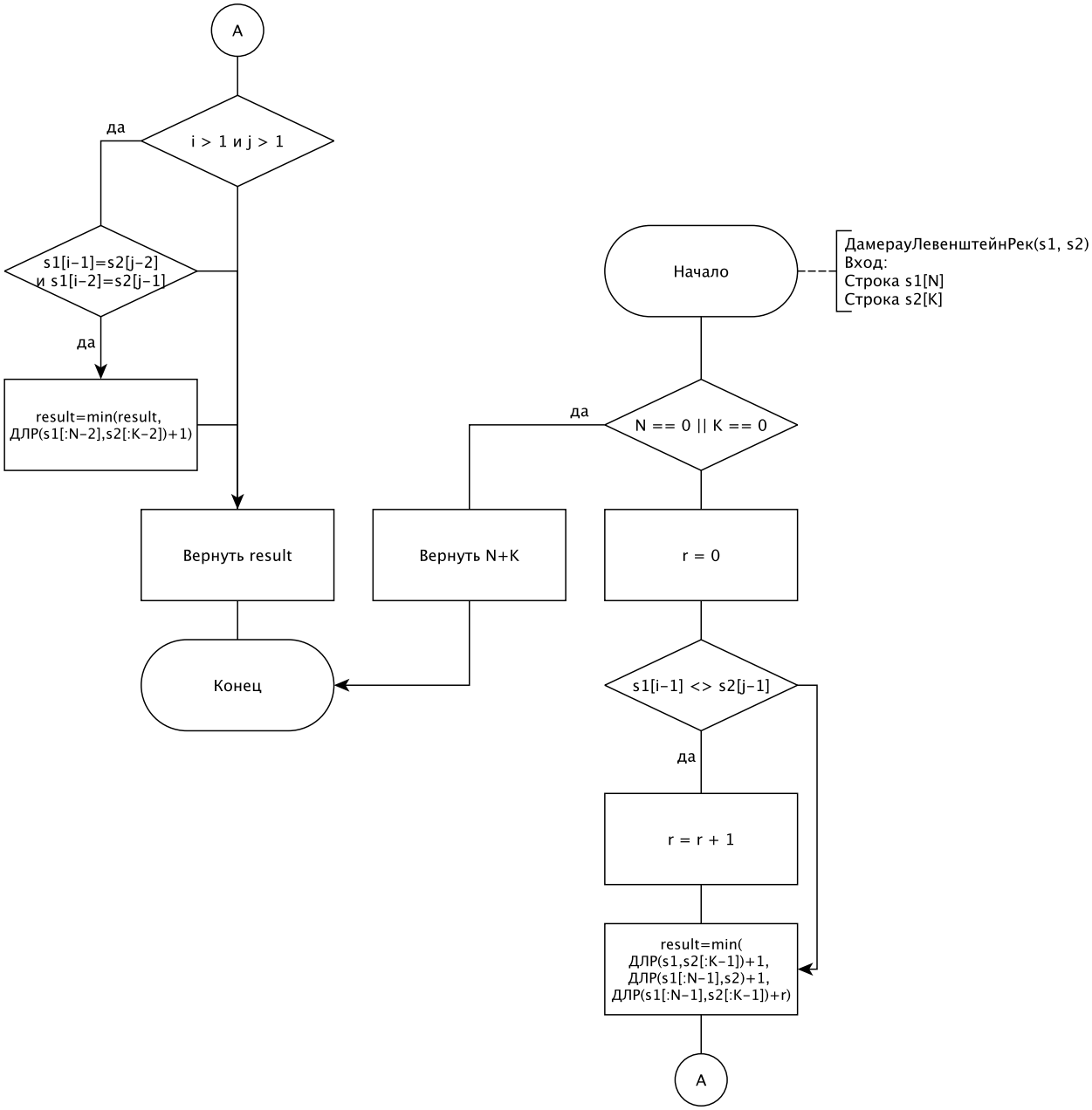

Рисунок 2.3 — Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

2.4. Разработка рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешем

На рисунке 2.4 представлена схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешем.

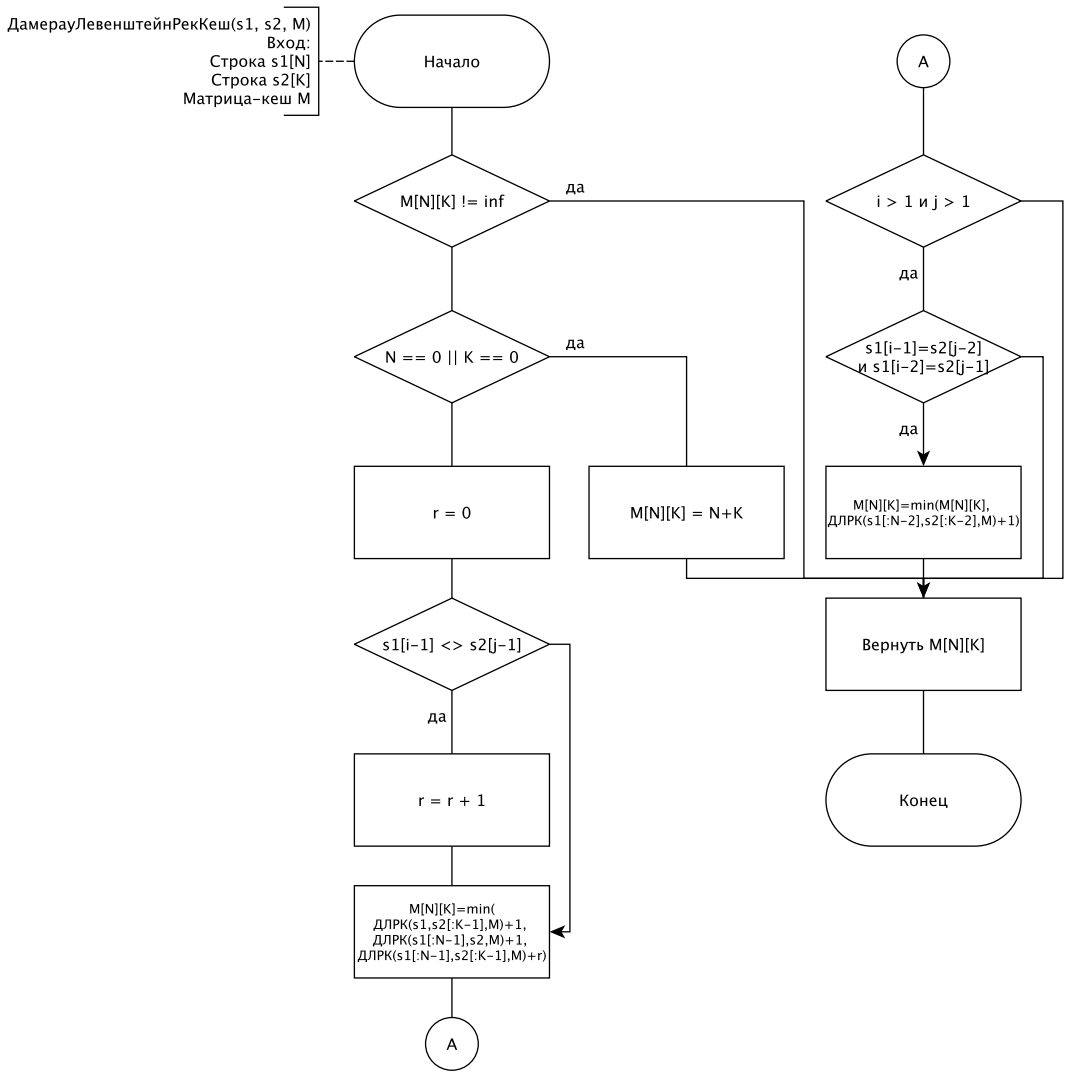

Рисунок 2.4 — Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешем

3. Технологическая часть

В данном разделе представлены реализации алгоритмов поиска редакционного расстояния. Кроме того, указаны требования к ПО и средства реализации.

3.1. Требования к ПО

— программа позволяет вводить строки с использованием аргументов командной строки;

— программа аварийно завершается в случае ошибок, выводя сообщение о соответствующей ошибке;

— программа выполняет замеры времени работы реализаций алгоритмов;

— программа строит зависимости времени работы реализаций алгоритмов от размеров входных данных;

— программа принимает на вход строки из символов различных языков.

3.2. Средства реализации

Для реализации данной работы выбран язык программирования Go, так как он содержит необходимые для тестирования библиотеки, а также обладает достаточными инструментами для реализации ПО, удовлетворяющего требованиям данной работы [3].

3.3. Реализации алгоритмов

В листингах 3.1 – 3.4 представлены реализации алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 — Исходный код реализации матричного алгоритма нахождения расстояния Левенштейна

```
func (l *Levenshtein) diff(s1, s2 []rune) int {
        m := make([][]int, len(s1)+1)
        for i := range m {
                m[i] = make([]int, len(s2)+1)
        }

        for i := range m[0] {
                m[0][i] = i
        }

        for i := range m {
                m[i][0] = i
        }

        for i := 1; i < len(m); i++ {
                for j := 1; j < len(m[i]); j++ {
                        r := 0
                        if s1[i-1] != s2[j-1] {
                                r++
                        }

                        m[i][j] = utils.Min(
                                m[i][j-1]+1,
                                m[i-1][j]+1,
                                m[i-1][j-1]+r,
                        )
                }
        }

        return m[len(m)-1][len(m[0])-1]
}
```

Листинг 3.2 — Исходный код реализации матричного алгоритма нахождения расстояния Дамерау-Левенштейна

```
func (l *DamerauLevenshtein) diff(s1, s2 []rune) int {
        m := make([][]int, len(s1)+1)
        for i := range m {
                m[i] = make([]int, len(s2)+1)
        }
        for i := range m[0] {
                m[0][i] = i
        }
        for i := range m {
                m[i][0] = i
        }
        for i := 1; i < len(m); i++ {
                for j := 1; j < len(m[i]); j++ {
                        r := 0
                        if s1[i-1] != s2[j-1] {
                                r++
                        }
                        m[i][j] = utils.Min(m[i][j-1]+1,m[i-1][j]+1,
                                m[i-1][j-1]+r,
                        )
                        if i > 1 && j > 1 {
                                if s1[i-1] == s2[j-2] && s1[i-2] ==
                                s2[j-1] {
                                        m[i][j] = utils.Min(m[i][j],
                                                m[i-2][j-2]+1,
                                        )
                                }
                        }
                }
        }
        return m[len(m)-1][len(m[0])-1]
}
```

Листинг 3.3 — Исходный код реализации рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

```
func (l *DamerauLevenshteinRecursive) diff(s1, s2 []rune) int {
        if len(s1) == 0 || len(s2) == 0 {
                return len(s1) + len(s2)
        }

        r := 0
        if s1[len(s1)-1] != s2[len(s2)-1] {
                r++
        }

        result := utils.Min(
                &l.stat,
                l.diff(s1, s2[:len(s2)-1])+1,
                l.diff(s1[:len(s1)-1], s2)+1,
                l.diff(s1[:len(s1)-1], s2[:len(s2)-1])+r,
        )

        if len(s1) > 1 && len(s2) > 1 {
                if s1[len(s1)-1] == s2[len(s2)-2] &&
                s1[len(s1)-2] == s2[len(s2)-1] {
                        result = utils.Min(
                                result,
                                l.diff(s1[:len(s1)-2], s2[:len(s2)-2])+1,
                        )
                }
        }

        return result
}
```

Листинг 3.4 — Исходный код реализации рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешем

```
func (l *DamerauLevenshteinRecursiveCached) diff1(s1, s2 []rune,
 cache [][]int) int {
        if cache[len(s1)][len(s2)] != math.MaxInt {
                return cache[len(s1)][len(s2)]
        }
        if len(s1) == 0 || len(s2) == 0 {
                cache[len(s1)][len(s2)] = len(s1) + len(s2)
        }
        r := 0
        if s1[len(s1)-1] != s2[len(s2)-1] {
                r++
        }
        cache[len(s1)][len(s2)] = utils.Min(
                l.diff1(s1, s2[:len(s2)-1], cache)+1,
                l.diff1(s1[:len(s1)-1], s2, cache)+1,
                l.diff1(s1[:len(s1)-1], s2[:len(s2)-1], cache)+r,
        )
        if len(s1) > 1 && len(s2) > 1 {
                if s1[len(s1)-1] == s2[len(s2)-2] &&
                 s1[len(s1)-2] == s2[len(s2)-1] {
                        cache[len(s1)][len(s2)] = utils.Min(
                                cache[len(s1)][len(s2)],
                                l.diff1(s1[:len(s1)-2],
                                s2[:len(s2)-2], cache)+1,
                        )
                }
        }

        return cache[len(s1)][len(s2)]
}
```

3.4. Тестирование

Тестирование проводилось по методологии чёрного ящика. Тесты пройдены успешно.

В таблице представлены тестовые данные для реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Таблица 3.1 — Тестовые данные для алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна

№	s_1	s_2	Левентшейн	Дамерау-Левенштейн
1	""	""	0	0
2	""	"a"	1	1
3	"a"	""	1	1
4	"bababa"	"bebebe"	3	3
5	"abc"	"cbc"	1	1
6	"опарышв"	"опарывш"	2	1
7	"ффф"	"енф"	2	2

4. Экспериментальная часть

В данном разделе описаны замерные эксперименты и представлены результаты исследования.

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялся эксперимент [5]:

— 8 ГБ оперативной памяти;

— процессор Apple M2;

— операционная система macOS Ventura 13.0.

4.2. Измерение процессорного времени выполнения реализаций алгоритмов

Для измерения процессорного времени выполнения реализаций алгоритмов была использована функция языка C — $clock\_gettime$, которая позволяет получить текущее процессорное время в наносекундах [6].

В таблице 4.1 представлены результаты измерений процессорного времени выполнения в зависимости от длины строки. На рисунке 4.1 представлена зависимость времени выполнения от длины строк.

Таблица 4.1 — Результаты замеров процессорного времени (в нс)

Длина	Лев.	Дам.-Лев.	Дам.-Лев. рек.	Дам.-Лев. рек. кеш.
1	480	360	440	730
2	650	710	1890	1730
3	1200	1230	8190	2960
4	1830	1650	87720	8030
5	3370	3330	289660	7500
6	2940	2870	929850	7670
7	2760	2760	4643940	9850
8	3630	3640	25635910	14060
10	5640	5630	784663680	20660

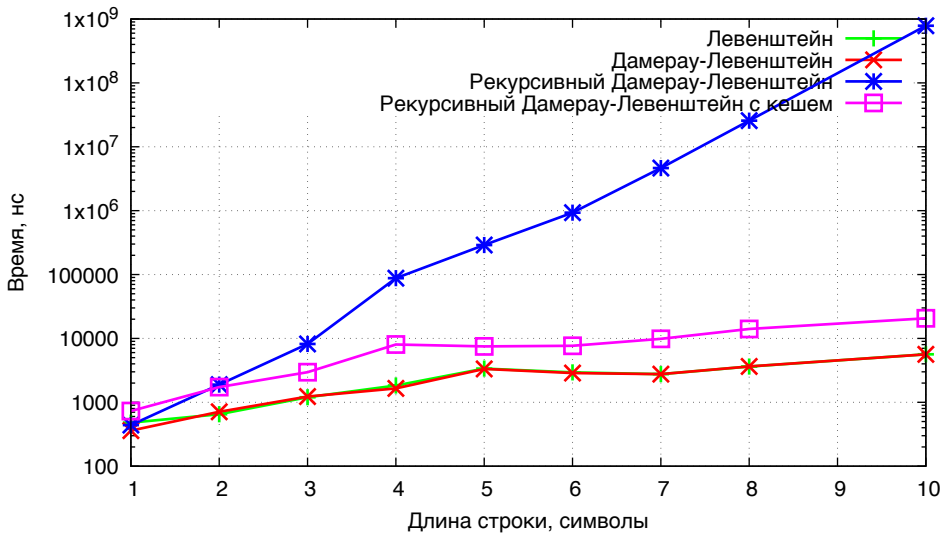

Рисунок 4.1 — Результаты замеров времени

Так как на этом масштабе не видна разница между реализациями матричных алгоритмов Левенштейна и Дамерау-Левенштейна, то на рисунке 4.2 представлена зависимость времени выполнения от длин строк для данных двух алгоритмов на более крупной выборке.

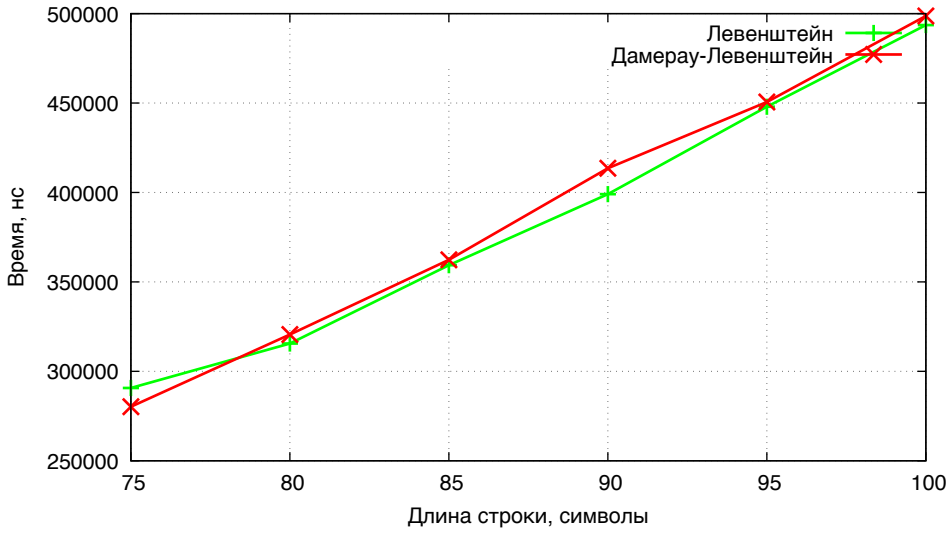

Рисунок 4.2 — Результаты замеров времени для матричных реализаций

4.3. Измерение объёма потребляемой памяти реализаций алгоритмов

В таблице 4.2 представлены результаты измерения потребляемой памяти в зависимости от длины строки. На рисунке 4.3 представлена зависимость потребляемой памяти от длины строк.

Таблица 4.2 — Результаты замеров потребляемой памяти (в байтах)

Длина	Лев (м)	Дам-Лев	Дам-Лев рек.	Дам-Лев рек. кеш.
1	252	252	144	356
2	281	281	288	561
3	312	312	432	768
4	345	345	576	977
5	380	380	720	1188
6	417	417	864	1401
7	456	456	1008	1616
8	497	497	1152	1833
10	585	585	1440	2273

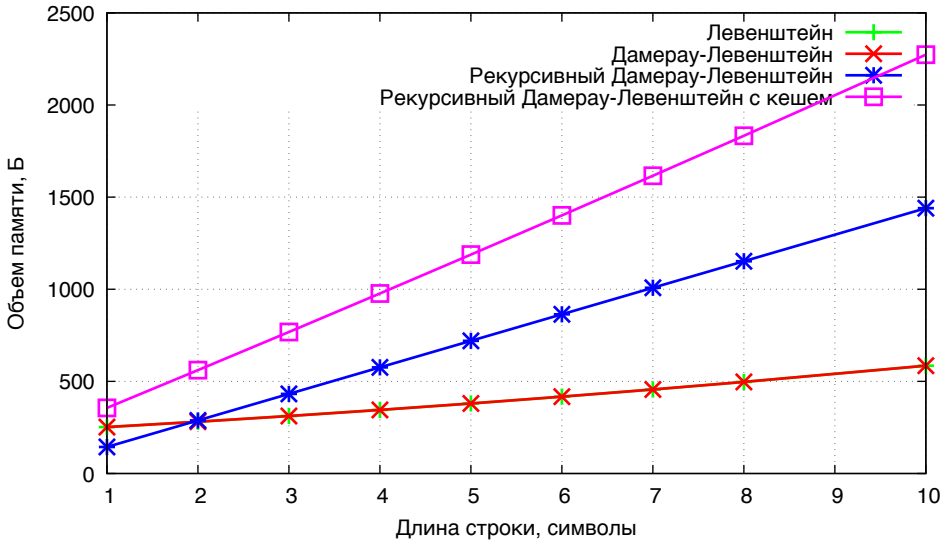

Рисунок 4.3 — Результаты замеров памяти

Заключение

Все дальнейшие выводы приведены исходя из анализов данных замеров для длин строк равных десяти, так как данная длина строки позволяет получить достаточно показательные результаты для проведения сравнения.

По итогам замеров можно сказать, что рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна в 139371 раз медленнее матричной реализации алгоритма.

Рекурсивная реализация с кешем медленнее матричной в 3 раза, но объем необходимой памяти в реализации с кешем в 3.8 раз больше, чем в матричной.

Матричный реализации алгоритма Левенштейна и Дамерау-Левенштейна не отличаются в плане использования памяти. Алгоритм Левенштейна быстрее алгоритма Дамерау-Левенштейна в 1.002 раза, так как они почти идентичны, с одним отличием — в алгоритме Дамерау-Левенштейна есть дополнительная проверка.

Цель работы была достигнута: были изучены алгоритмы поиска редакционного расстояния. Были выполнены все задачи:

— изучены алгоритмы вычисления редакционного расстояния;

— кодированы данные алгоритмы;

— проведен замерный эксперимент для данных алгоритмов, с измерением времени работы и использования памяти;

— проведен сравнительный анализ результатов полученных данных.

Список использованных источников

[1] Погорелов Д. А., Таразанов А. М., Волкова Л. Л. СРАВНИТЕЛЬНЫЙ АНАЛИЗ АЛГОРИТМОВ РЕДАКЦИОННОГО РАССТОЯНИЯ ЛЕВЕНШТЕЙНА И ДАМЕРАУ-ЛЕВЕНШТЕЙНА //Синергия Наук. – 2019. – №. 31. – С. 803-811.

[2] Зубков С. Assembler. Для DOS, Windows и Unix. – Litres, 2022.

[3] Документация по языку программирования *Go* [Электронный ресурс]. Режим доступа: *https://go.dev/doc* (дата обращения: 07.10.2022).

[4] Документация по пакетам языка программирования *Go* [Электронный ресурс]. Режим доступа: *https://pkg.go.dev* (дата обращения: 07.10.2022).

[5] Техническая спецификация ноутбука *MacBookAir* [Электронный ресурс]. Режим доступа: *https://support.apple.com/kb/SP869* (дата обращения: 08.10.2022).

[6] Документация по функции *clock_gettime* [Электронный ресурс]. Режим доступа: *https://man7.org/linux/man-pages/man3/clock_gettime.3.html* (дата обращения: 25.10.2022).