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 дисциплине «Анализ алгоритмов»

Тема: Редакционные расстояния
Студент: Карпова Е. О.
Группа: ИУ7-52Б
Оценка (баллы): _____
Преподаватели: Волкова Л. Л., Строганов Ю. В.

Москва — 2022 г.

Оглавление

Введение

Расстояние Левенштейна (редакционное расстояние) — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой (редакционных операций), необходимых для превращения одной последовательности символов в другую.

Редакционные операции можно обозначить следующим образом:

- i (insert) — вставка одного символа;
- d (delete) — удаление одного символа;
- r (replace) — замена одного символа на другой.

Расстояние Дамерау – Левенштейна — это минимальное количество операций вставки одного символа, удаления одного символа, замены одного символа на другой и транспозиции двух соседних символов (редакционных операций), необходимых для превращения одной последовательности символов в другую. Можно назвать модифицированным расстоянием Левенштейна.

Таким образом, вводится одно обозначение в дополнение к перечисленным выше:
t (transposition) — транспозиция двух соседних символов.

Расстояние Левенштейна и расстояние Дамерау – Левенштейна находят применение в:

- компьютерной лингвистике;
- биоинформатике (сравнение генов);
- базах данных [1];
- распознавании рукописных символов [1].

Цель работы: получение навыков программирования, тестирования полученного программного продукта и проведения замеров времени выполнения и потребляемой памяти по результатам работы программы на примере решения задачи о редакционном расстоянии.

Задачи работы:

1) изучение расстояний Левенштейна и Дамерау – Левенштейна;
2) изучение алгоритма вычисления расстояния Левенштейна, Дамерау – Левенштейна, рекурсивного алгоритма вычисления расстояния Дамерау – Левенштейна и модификации последнего с использованием кэша;
3) разработка нерекурсивных алгоритмов поиска расстояний Левенштейна и Дамерау – Левенштейна, рекурсивного алгоритм поиска расстояния Дамерау – Левенштейна и модификации последнего с использованием кэша;
4) реализация данных алгоритмов;
5) проведение замеров потребления памяти (в байтах) для данных алгоритмов;
6) проведение замеров времени работы (в нс) данных алгоритмов;
7) получение графической зависимости замеряемых величин от длины последовательности символов, предоставляемой на вход алгоритмам;
8) проведение сравнительного анализа данных алгоритмов на основе полученных зависимостей.

1. Аналитическая часть

В данном разделе будут рассмотрены теоретические основы алгоритмов поиска редакционного расстояния и общие сведения по этой задаче.

За редакционное расстояние принимают минимальное количество операций над строкой, с помощью которых она может быть преобразована в другую. Поэтому для каждой операции над строкой вводится условная цена. Суммарная стоимость всех произведённых операций и будет являться редакционным расстоянием между данными строками.

Цены за редакционные операции можно обозначить и определить следующим образом:

- $\omega(\alpha, \beta) = 1$ — цена замены при $\alpha \neq \beta$;
- $\omega(\alpha, \alpha) = 0$ — цена эквивалентной замены;
- $\omega(\varnothing, \beta) = 1$ — цена вставки;
- $\omega(\alpha, \varnothing) = 1$ — цена удаления.

Для описания алгоритма вычисления расстояния Левенштейна поиска редакционного расстояния применяется рекуррентное (т.е. использующее результаты предшествующих вычислений на следующем шаге) соотношение

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

Для алгоритма вычисления расстояния Дамерау – Левенштейна (1.1) примет вид

$$D(s_1[1..i], s_2[1..j]) = \begin{cases} 0, & \text{если } i = 0, j = 0, \\ i, & \text{если } i > 0, j = 0, \\ j, & \text{если } i = 0, j > 0, \\ \min(& D(s_1[1..i], s_2[1..j-1]) + 1, \\ & D(s_1[1..i-1], s_2[1..j]) + 1, \\ & D(s_1[1..i-1], s_2[1..j-1]) + m, \\ & D(s_1[1..i-2], s_2[1..j-2]) + n, \qquad \text{если } i > 1, j > 1 \\ ). \end{cases} \tag{1.2}$$

При этом

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

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

Из этого следует, что для реализации алгоритмов могут быть использованы два подхода: матричный и рекурсивный.

Под рекурсивным подходом понимается реализация, при которой значения предыдущих членов, необходимых для вычисления редакционного расстояния, высчитываются на каждом вызове.

Под матричным подходом понимается реализация, при которой значения предыдущих членов хранятся в матрице.

1.1. Матричный алгоритм нахождения расстояния Левенштейна

Если реализовывать вычисления по формуле 1.1 рекурсивно напрямую, то при больших размерах строк эффективность работы программы будет ниже. Это связано с тем, что в определённых местах вычисления, которые уже производились на предыдущих шагах, будут производиться повторно.

Матричный алгоритм подразумевает, что в памяти хранится вся матрица результатов (M), что позволит при совпадении начальных значений для вычислений на текущем шаге не выполнять их заново, а найти готовый результат в соответствующей ячейке матрицы. В матрице текущему значению $D(s_1[1..i], s_2[1..j])$ будет соответствовать ячейка $M[i][j]$.

1.2. Матричный алгоритм нахождения расстояния Дамерау – Левенштейна

Рассуждения аналогичны пункту 1.1., но используется (1.2). Это связано с тем, что в данном алгоритме предусмотрена ещё одна редакционная операция — транспозиция двух соседних символов.

1.3. Рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна

Рекурсивный алгоритм поиска редакционного расстояния Дамерау – Левенштейна реализует (1.2).

Пусть $s_1^{'}$ — строка s_1 без последнего символа, а $s_2^{'}$ — строка s_2 без последнего символа. Цену преобразования из строки s_1 в строку s_2 обозначим как $\omega(s_1, s_2)$.

За известные принимаются и используются для составления формулы следующие утверждения:

- цена преобразования между двумя пустыми строками равняется 0;
- цена преобразования между непустой строкой и пустой строкой равняется длине непустой строки (вне зависимости от порядка операндов);
- цена преобразования строки s_1 в строку s_2 может быть выражена как $\omega(s_1, s_2^{'}) + \omega(s_2^{'}, s_2)$, где второе слагаемое равняется $\omega(\varnothing, \beta) = 1$ — цена вставки, где β — любой символ;
- цена преобразования строки s_1 в строку s_2 может быть выражена как $\omega(s_1^{'}, s_2) + \omega(s_1, s_1^{'})$, где второе слагаемое равняется $\omega(\alpha, \varnothing) = 1$ — цена удаления, где α — любой символ;
- цена преобразования строки s_1 в строку s_2 может быть выражена как $\omega(s_1^{'}, s_2^{'}) + \omega(\alpha, \beta)$, где α, β — последние символы строк s_1, s_2 соответственно, а второе слагаемое равняется 1 при $\alpha \neq \beta$ и равняется 0 иначе;

- цена преобразования строки s_1 в строку s_2, если их длина превышает 2, может быть выражена как $\omega((s_1')', (s_2')') + \omega(\alpha\beta, \delta\gamma)$, где комбинации $(\alpha\beta)$, $(\delta\gamma)$ — последние два символа строк s_1, s_2 соответственно, а второе слагаемое равняется 1 при $\alpha = \gamma$ и $\beta = \delta$ и равняется 0 при $\alpha = \delta$ и $\beta = \gamma$.

За редакционное расстояние на текущем шаге принимается минимальное из перечисленных значений.

Такой подход является невыгодным с точки зрения потребляемой памяти при рекурсивных вызовах и из-за роста числа повторных вычислений.

1.4. Рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна с кэшированием

Рекурсивную реализацию алгоритма нахождения редакционного расстояния можно оптимизировать по времени выполнения с использованием кэша вычисленных значений в виде матрицы, заполняемой по правилам из матричной реализации в пункте 1.2.

Перед началом выполнения алгоритма все значения матрицы заполняются ∞. Элемент матрицы будет рассматриваться как ранее вычисленный, если в нём будет находиться значение, отличное от ∞, и не вычисленным ранее, иначе.

Таким образом, при вызове рекурсии вычисленные ранее значения не будут пересчитываться, а будут взяты из кэша, что значительно сократит время работы алгоритма.

2. Конструкторская часть

В данном разделе будут представлены схемы алгоритмов поиска редакционного расстояния, среди которых матричная реализация алгоритма вычисления расстояния Левенштейна и три реализации алгоритма вычисления расстояния Дамерау – Левенштейна: матричная, рекурсивная и рекурсивная с кэшированием.

2.1. Разработка матричной реализации алгоритма нахождения расстояния Левенштейна

На рисунках 2.1 – 2.2 представлена схема матричного алгоритма нахождения расстояния Левенштейна.

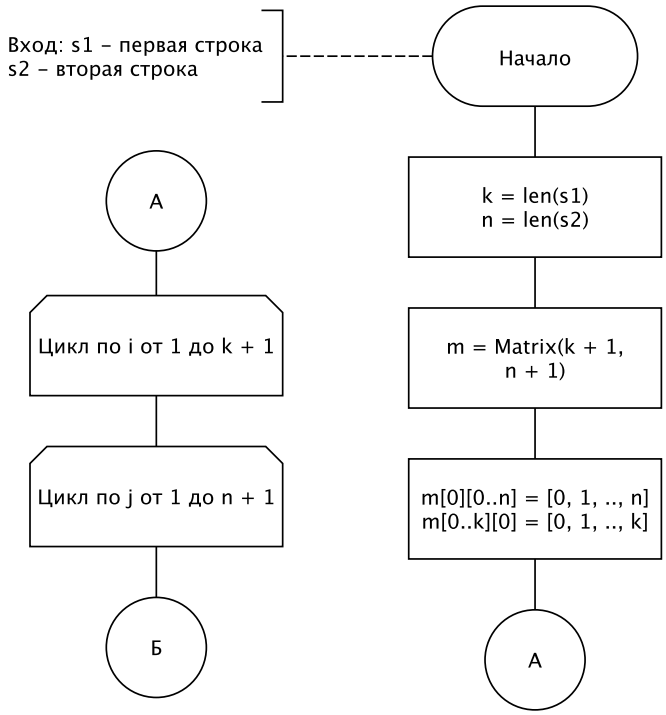

Рисунок 2.1 — Схема матричной реализации алгоритма нахождения расстояния Левенштейна

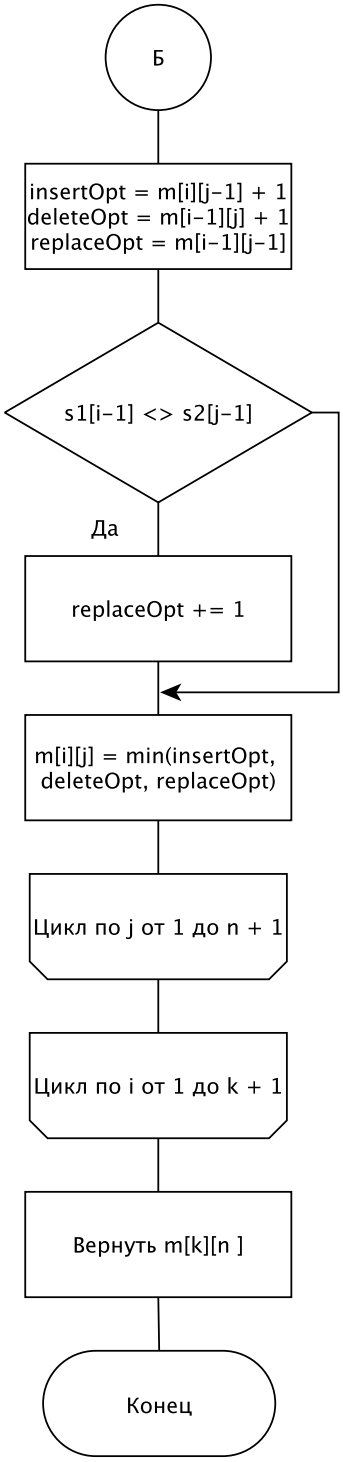

Рисунок 2.2 — Схема матричной реализации нахождения расстояния Левенштейна (продолжение рис. 2.1)

2.2. Разработка матричной реализации алгоритма нахождения расстояния Дамерау – Левенштейна

На рисунке 2.3 представлена схема матричной реализации алгоритма нахождения расстояния Дамерау – Левенштейна.

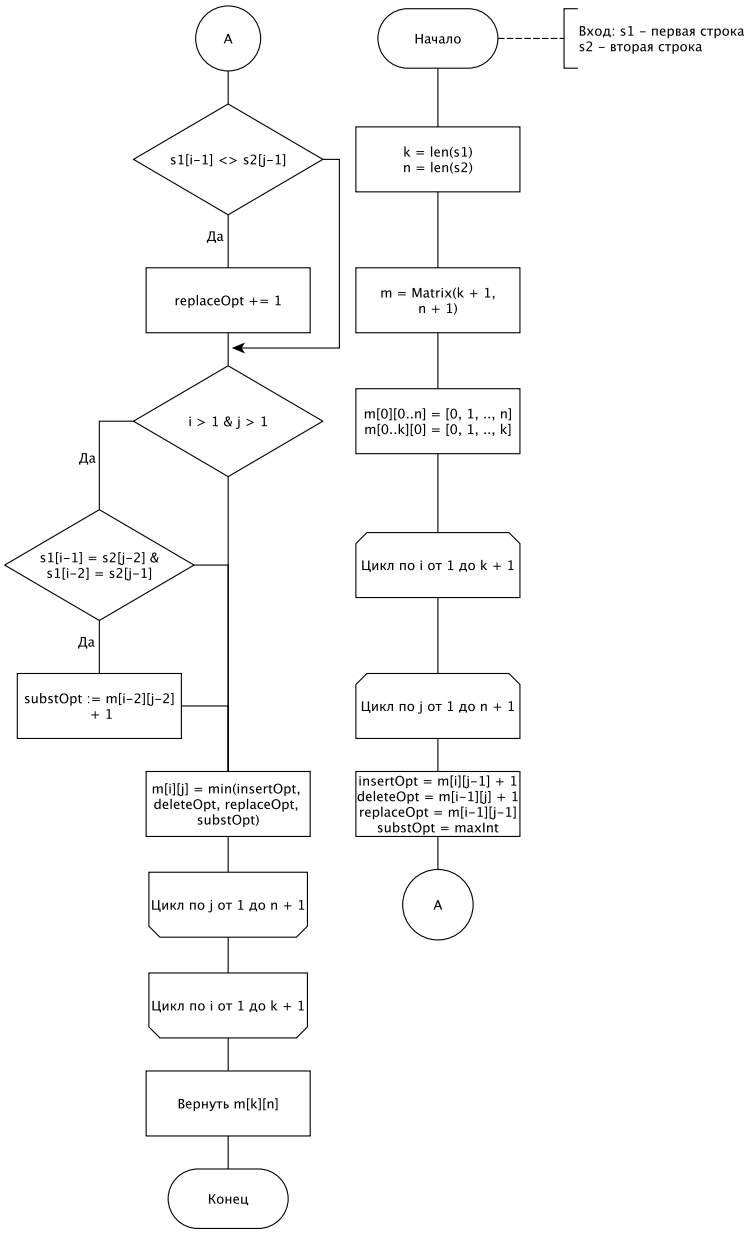

Рисунок 2.3 — Схема матричной реализации алгоритма нахождения расстояния Дамерау – Левенштейна

2.3. Разработка рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна

На рисунке 2.4 представлена схема рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна.

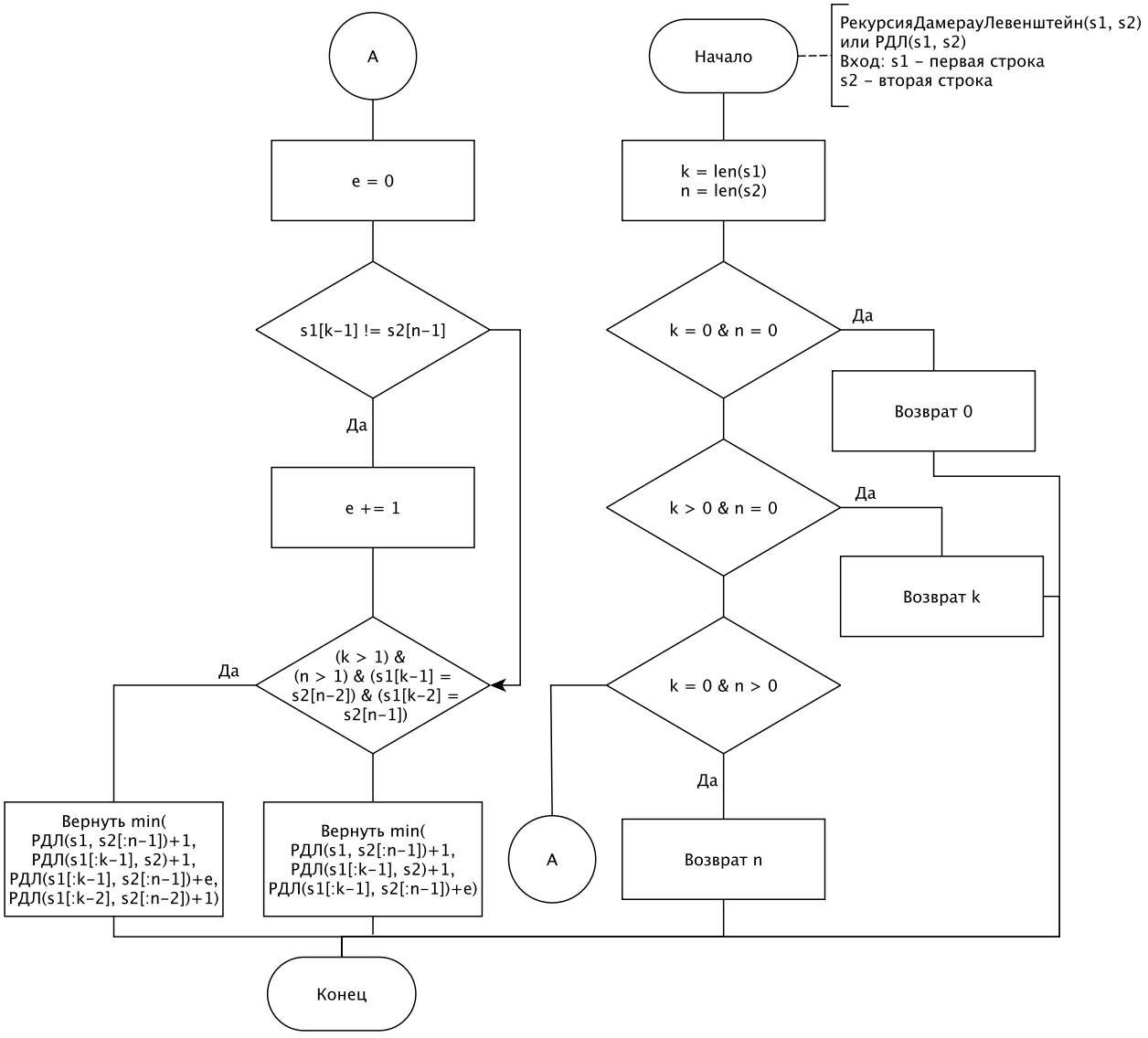

Рисунок 2.4 — Схема рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна

2.4. Разработка рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна с кэшем

На рисунке 2.5 представлена схема рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна с кэшем.

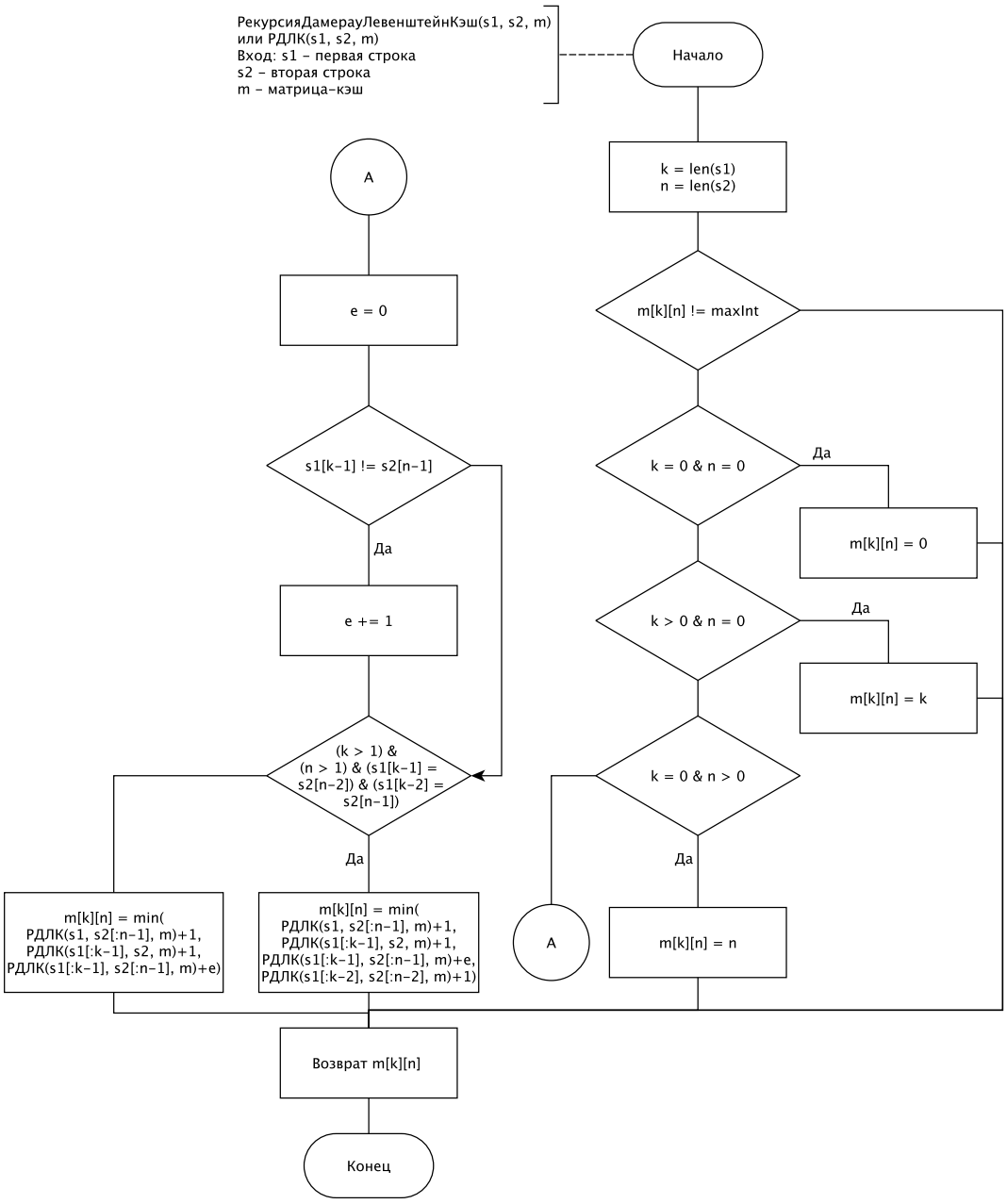

Рисунок 2.5 — Схема рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна с кэшем

3. Технологическая часть

В данном разделе будет представлена реализация алгоритмов поиска редакционного расстояния. Также будут указаны обязательные требования к ПО, средства реализации алгоритмов и результаты проведённого тестирования программы.

3.1. Требования к ПО

Для программы выделен перечень требований:

- предоставляется интерфейс в формате меню с возможностью ввода и изменения обрабатываемых строк, завершения работы с программой и выбора используемого для обработки введённых строк алгоритма поиска редакционного расстояния;
- предлагается повторно выполнить ввод при невалидном выборе пункта меню;
- производится аварийное завершение с текстом об ошибке при иных ошибках;
- проводится модульное тестирование функций реализующих поиск редакционного расстояния;
- производятся замеры времени выполнения и потребления памяти функциями поиска редакционного расстояния;
- допускается ввод строк в любой раскладке.

3.2. Средства реализации

Для реализации данной работы был выбран язык программирования Go [2]. Выбор обусловлен наличием в *Go* библиотек для тестирования ПО и проведения замеров времени выполнения в наносекундах, а также необходимых для реализации поставленных цели и задач средств. В качестве среды разработки была выбрана *GoLand* [4].

3.3. Реализация алгоритмов

В листингах 3.1 – 3.8 представлены различные реализации алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна.

Листинг 3.1 — Листинг матричной реализации алгоритма нахождения расстояния Левенштейна

```
func Levenshtein(s1, s2 string) int {
        s1Rune := []rune(s1)
        s2Rune := []rune(s2)
        m := make([][]int, len(s1Rune)+1)
        for i := range m {
                m[i] = make([]int, len(s2Rune)+1)
                m[i][0] = i
        }
        for j := range m[0] {
                m[0][j] = j
        }

        for i := 1; i < len(m); i++ {
                for j := 1; j < len(m[i]); j++ {
                        insertOpt := m[i][j-1] + 1
                        deleteOpt := m[i-1][j] + 1
                        replaceOpt := m[i-1][j-1]

                        if s1Rune[i-1] != s2Rune[j-1] {
                                replaceOpt += 1
                        }
                        m[i][j] = min(insertOpt, deleteOpt, replaceOpt)
                }
        }
        if Print {
                PrintMatrix(m)
        }

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

Листинг 3.2 — Листинг матричной реализации алгоритма нахождения расстояния Дамерау – Левенштейна (начало)

```
func DamerauLevenshtein(s1, s2 string) int {
        s1Rune := []rune(s1)
        s2Rune := []rune(s2)

        m := make([][]int, len(s1Rune)+1)
        for i := range m {
                m[i] = make([]int, len(s2Rune)+1)
                m[i][0] = i
        }
        for j := range m[0] {
                m[0][j] = j
        }

        for i := 1; i < len(m); i++ {
                for j := 1; j < len(m[i]); j++ {
                        insertOpt := m[i][j-1] + 1
                        deleteOpt := m[i-1][j] + 1
                        replaceOpt := m[i-1][j-1]
                        substituteOpt := math.MaxInt

                        if s1Rune[i-1] != s2Rune[j-1] {
                                replaceOpt += 1
                        }

                        if i > 1 && j > 1 {
                                if s1Rune[i-1] == s2Rune[j-2] &&
                                s1Rune[i-2] == s2Rune[j-1] {
                                        substituteOpt = m[i-2][j-2] + 1
                                }
                        }
```

Листинг 3.3 — Листинг матричной реализации алгоритма нахождения расстояния Дамерау – Левенштейна (окончание листинга 3.2)

```
                        m[i][j] = min(insertOpt, deleteOpt, replaceOpt,
                        substituteOpt)
                }
        }

        if Print {
                PrintMatrix(m)
        }

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

Листинг 3.4 — Листинг рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна (начало)

```
func RecursiveDamerauLevenshtein(s1, s2 string) int {
        s1Rune := []rune(s1)
        s2Rune := []rune(s2)

        return recursiveDamerauLevenshtein(s1Rune, s2Rune)
}

func recursiveDamerauLevenshtein(s1, s2 []rune) int {

        if len(s1) == 0 && len(s2) == 0 {
                return 0
        } else if len(s1) > 0 && len(s2) == 0 {
                return len(s1)
        } else if len(s1) == 0 && len(s2) > 0 {
                return len(s2)
        } else {
                e := 0
```

Листинг 3.5 — Листинг рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна (окончание листинга 3.4)

```
                if s1[len(s1)-1] != s2[len(s2)-1] {
                        e += 1
                }

                if len(s1) > 1 && len(s2) > 1 &&
                        (s1[len(s1)-1] == s2[len(s2)-2] &&
                        s1[len(s1)-2] == s2[len(s2)-1]) {
                        return min(recursiveDamerauLevenshtein(s1,
                        s2[:len(s2)-1])+1,
                                recursiveDamerauLevenshtein(s1[:len(s1)-1],
                                s2)+1,
                                recursiveDamerauLevenshtein(s1[:len(s1)-1],
                                s2[:len(s2)-1])+e,
                                recursiveDamerauLevenshtein(s1[:len(s1)-2],
                                s2[:len(s2)-2])+1)
                } else {
                        return min(recursiveDamerauLevenshtein(s1,
                        s2[:len(s2)-1])+1,
                                recursiveDamerauLevenshtein(s1[:len(s1)-1],
                                s2)+1,
                                recursiveDamerauLevenshtein(s1[:len(s1)-1],
                                s2[:len(s2)-1])+e)
                }
        }
}
```

Листинг 3.6 — Листинг рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна с кэшем (начало)

```
func RecursiveDamerauLevenshteinCached(s1, s2 string) int {
        s1Rune := []rune(s1)
        s2Rune := []rune(s2)

        cache := make([][]int, len(s1)+1)
        for i := range cache {
                cache[i] = make([]int, len(s2)+1)
                cache[i][0] = i
        }
        for j := range cache[0] {
                cache[0][j] = j
        }

        for i := 1; i < len(cache); i++ {
                for j := 1; j < len(cache[i]); j++ {
                        cache[i][j] = math.MaxInt
                }
        }

        res := recursiveDamerauLevenshteinCached(s1Rune, s2Rune, cache)

        if Print {
                PrintMatrix(cache)
        }

        return res
}

func recursiveDamerauLevenshteinCached(s1, s2 []rune, cache [][]int) int {
        if cache[len(s1)][len(s2)] == math.MaxInt {
                if len(s1) == 0 && len(s2) == 0 {
```

Листинг 3.7 — Листинг рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна с кэшем (продолжение листинга 3.6)

```
                cache[len(s1)][len(s2)] = 0
        } else if len(s1) > 0 && len(s2) == 0 {
                cache[len(s1)][len(s2)] = len(s1)
        } else if len(s1) == 0 && len(s2) > 0 {
                cache[len(s1)][len(s2)] = len(s2)
        } else {
                e := 0
                if s1[len(s1)-1] != s2[len(s2)-1] {
                        e += 1
                }
                if len(s1) > 1 && len(s2) > 1 &&
                        (s1[len(s1)-1] == s2[len(s2)-2] &&
                        s1[len(s1)-2] == s2[len(s2)-1]) {
                        cache[len(s1)][len(s2)] = min(
                            recursiveDamerauLevenshteinCached(
                            s1, s2[:len(s2)-1], cache)+1,
                                recursiveDamerauLevenshteinCached(
                                s1[:len(s1)-1], s2, cache)+1,
                                recursiveDamerauLevenshteinCached(
                                s1[:len(s1)-1], s2[:len(s2)-1],
                                cache)+e,
                                recursiveDamerauLevenshteinCached(
                                s1[:len(s1)-2], s2[:len(s2)-2],
                                cache)+1)
                } else {
                        cache[len(s1)][len(s2)] = min(
                            recursiveDamerauLevenshteinCached(
                            s1, s2[:len(s2)-1], cache)+1,
                                recursiveDamerauLevenshteinCached(
                                s1[:len(s1)-1], s2, cache)+1,
                                recursiveDamerauLevenshteinCached(
                                s1[:len(s1)-1], s2[:len(s2)-1],
                                cache)+e)
```

Листинг 3.8 — Листинг рекурсивной реализации алгоритма нахождения расстояния Дамерау – Левенштейна с кэшем (окончание листинга 3.7)

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

3.4. Тестирование

В таблице представлены тесты для алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна. Тестирование проводилось по методологии чёрного ящика. Все тесты пройдены успешно.

Таблица 3.1 — Тесты для алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна

№	s_1	s_2	Ожидаемый результат	
			Левентшейн	Дамерау – Левенштейн
1	∅	∅	0	0
2	abc	aba	1	1
3	text	tetx	2	1
4	скат	кот	2	2
5	∅	ababab	6	6
6	асаса	∅	5	5
7	∅	пюрешка	7	7
8	сосиска	∅	7	7
9	кеотон	кетоно	2	2
10	мышь	мішь	1	1
11	кушнявка	укшняква	4	2
12	привет	рпвите	4	3

4. Экспериментальная часть

В данном разделе описаны проведённые замеры и представлены результаты исследования. Также будут уточнены характеристики устройства, на котором проводились замеры времени выполнения и потребляемой памяти.

4.1. Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование [5]:

- операционная система macOS Monterey 12.4;
- 8 ГБ оперативной памяти;
- процессор Apple M2 (базовая частота — 2400 МГц, но поддержка технологии Turbo Boost позволяет достигать частоты в 3500 МГц [6]).

4.2. Измерение времени выполнения реализаций алгоритмов

Тестирование реализаций алгоритмов производилось при помощи встроенных в Go средств, а именно «бенчмарков» (*benchmarks* из пакета *testing* стандартной библиотеки *Go* [3]), представляющих собой тесты производительности. По умолчанию производятся только замеры процессорного времени выполнения в наносекундах [7][8], но при добавлении ключа $-benchmem$ также выполняется замер потребления памяти и количества аллокаций памяти.

Значение N динамически изменяется для достижения стабильного результата при различных условиях, но гарантируется, что каждый «бенчмарк» будет выполняться хотя бы одну секунду. Для замеров использовались строки равной длины, генерирующиеся случайным образом из строчных и прописных букв латинского алфавита перед началом выполнения «бенчмарков». Результаты тестирования возвращаются в структуре специального вида. Пример такой структуры представлен в листинге 4.9.

Листинг 4.9 — Листинг структуры результата «бенчмарка»

```
testing.BenchmarkResult{N:120000, T:1200000000, Bytes:0, MemAllocs:0x0,
MemBytes:0x0, Extra:map[string]float64{}}
```

В листинге 4.10 представлен пример реализации «бенчмарка», где $alg.function$ — объект типа функция (в данной реализации — функция, описывающая один из алгоритмов поиска редакционного расстояния).

Листинг 4.10 — Листинг примера реализации «бенчмарка»

```
func NewBenchmark(s1, s2 string, alg algorithm) func(*testing.B) {
        return func(b *testing.B) {
                for j := 0; j < b.N; j++ {
                        alg.function(s1, s2)
                }
        }
}
```

Результаты замеров времени выполнения (в нс.) приведены в таблице 4.1. Сокращение Д – Л обозначает алгоритм вычисления расстояния Дамерау – Левенштейна, (м) — матричная реализация, (р) — рекурсивная, (рк) — рекурсивная с кэшем. В таблице для значений, для которых замеры не выполнялись, в поле результата находится «—». Замеры времени для реализации рекурсивного алгоритма без кэша выполнены для первых семи значений длин строк, полученные результаты показывают различия в характере роста затрачиваемого реализациями времени: наибольшая скорость роста наблюдается для рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна без кэша, что подтверждает теоретическую оценку быстродействия (см. п. 1.1.).

Таблица 4.1 — Результаты замеров времени (нс.)

Длина строк	Левентшейн (м)	Д – Л (м)	Д – Л (р)	Д – Л (рк)
1	67	66	21	72
2	102	107	66	126
3	136	144	261	188
5	244	268	6600	402
6	297	325	35 055	545
7	350	388	189 077	684
10	614	703	31 517 684	1319
15	1229	1463	—	2782
20	2162	2565	—	4934
30	5031	5820	—	10 738
50	14 009	15 811	—	35 395

На рисунке 4.1 приведен график, отображающий зависимость времени работы реализаций алгоритмов от длин строк для всех реализаций алгоритмов, кроме рекурсивной реализации алгоритма поиска расстояния Дамерау–Левенштейна.

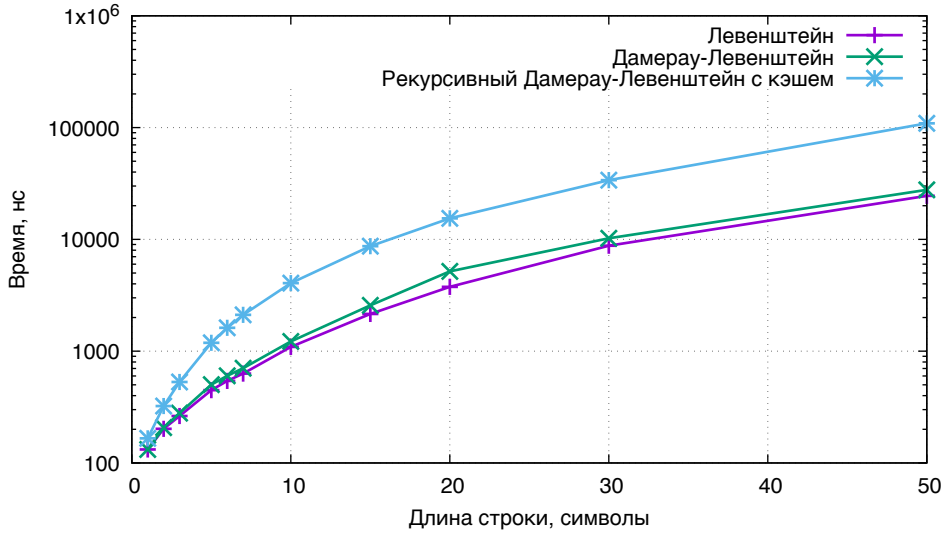

Рисунок 4.1 — Зависимость времени работы реализаций алгоритмов вычисления редакционного расстояния от длины строк

На рисунке 4.2 приведен график, отображающий зависимость времени работы реализаций алгоритмов от длин строк для всех реализаций алгоритмов при входных строках с длиной не более 10 символов.

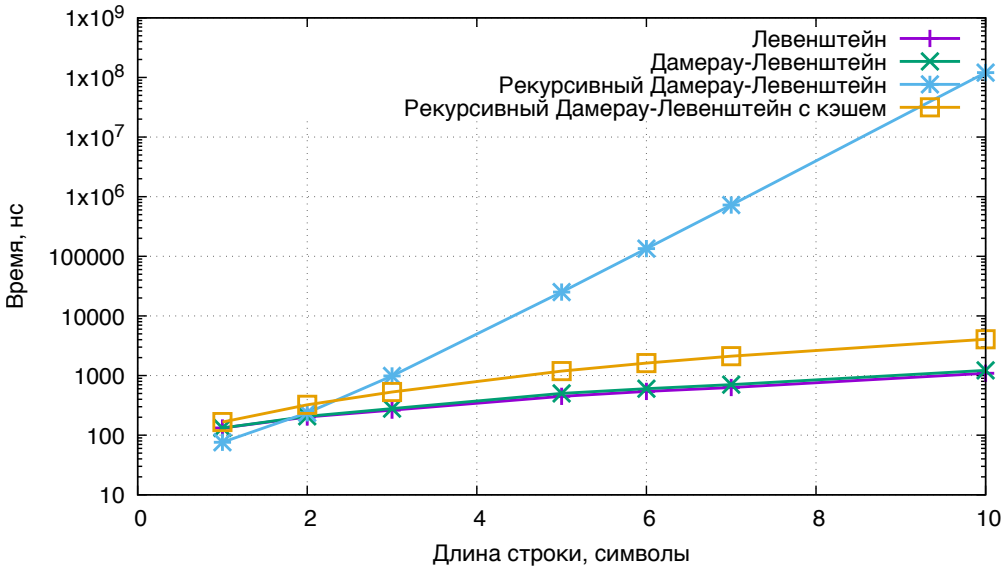

Рисунок 4.2 — Зависимость времени работы реализаций алгоритмов вычисления редакционного расстояния от длины строк (не более 10 символов)

4.3. Измерение объёма потребляемой реализациями алгоритмов памяти

Измерение объёма потребляемой памяти производилось посредством создания собственного программного модуля memory, использующего функции пакета unsafe языка *Go* [3]. Реализация основного функционала модуля, а также пример функции расчёта памяти, затрачиваемой на алгоритм поиска расстояния Левенштейна, и пример использования указанных функций приведены в Приложении А.

Результаты замеров потребляемой памяти (в байтах) приведены в таблице 4.2. Сокращение Д – Л обозначает алгоритм вычисления расстояния Дамерау – Левенштейна, (м) — матричная реализация, (р) — рекурсивная, (рк) — рекурсивная с кэшем. В таблице для значений, для которых тестирование не выполнялось, в поле результата находится «—». Замеры памяти для реализации рекурсивного алгоритма без кэша выполнены для пер-

вых семи значений длин строк, полученные результаты показывают различия в характере роста потребляемой реализациями памяти: наибольшая скорость роста наблюдается для рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна с кэшем, что подтверждает теоретическую оценку потребления памяти (см. п. 1.4.).

Таблица 4.2 — Результаты замеров потребляемой памяти (в байтах)

Длина строк	Левентшейн (м)	Д – Л (м)	Д – Л (р)	Д – Л (рк)
1	330	346	418	602
2	404	420	748	1044
3	494	510	1078	1502
5	722	738	1738	2466
6	860	876	2068	2972
7	1014	1030	2398	3494
10	1572	1588	3388	5156
15	2822	2838	—	8246
20	4472	4488	—	11 736
30	8972	8988	—	19 916
50	22 772	22 788	—	41 076

На рисунке 4.3 приведён график, отображающий зависимость потребляемой памяти от длин строк для реализаций алгоритмов поиска расстояния Левенштейна и Дамерау – Левенштейна на длинах строк не более 10 символов, так как результаты замеров памяти схожи, что не позволяет увидеть различия при большем масштабе графика. На рисунке 4.4 приведён график, отображающий зависимость потребляемой памяти от длин строк для всех алгоритмов на длинах строк не более 10 символов.

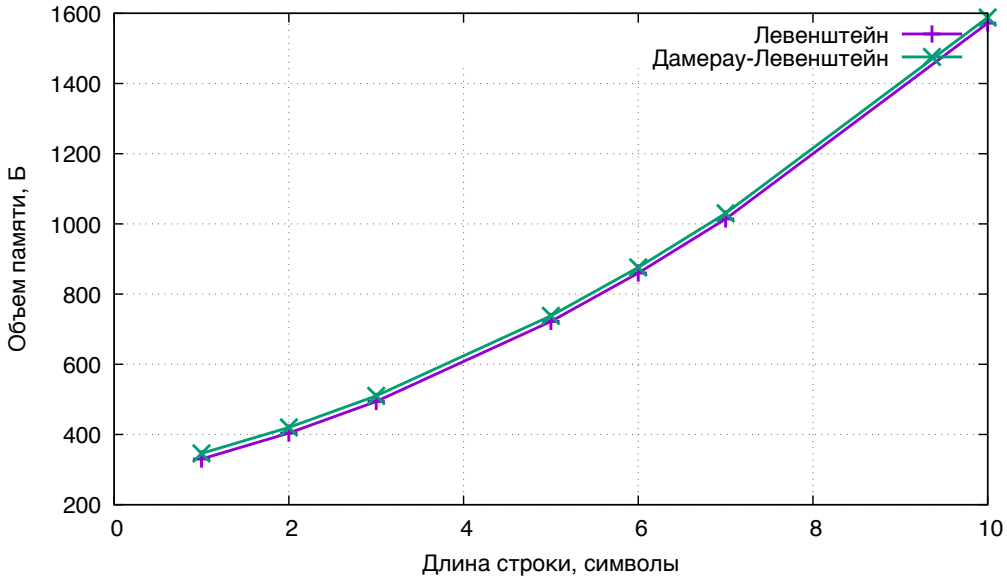

Рисунок 4.3 — Зависимость потребляемой памяти при вычислении редакционного расстояния от длины строк

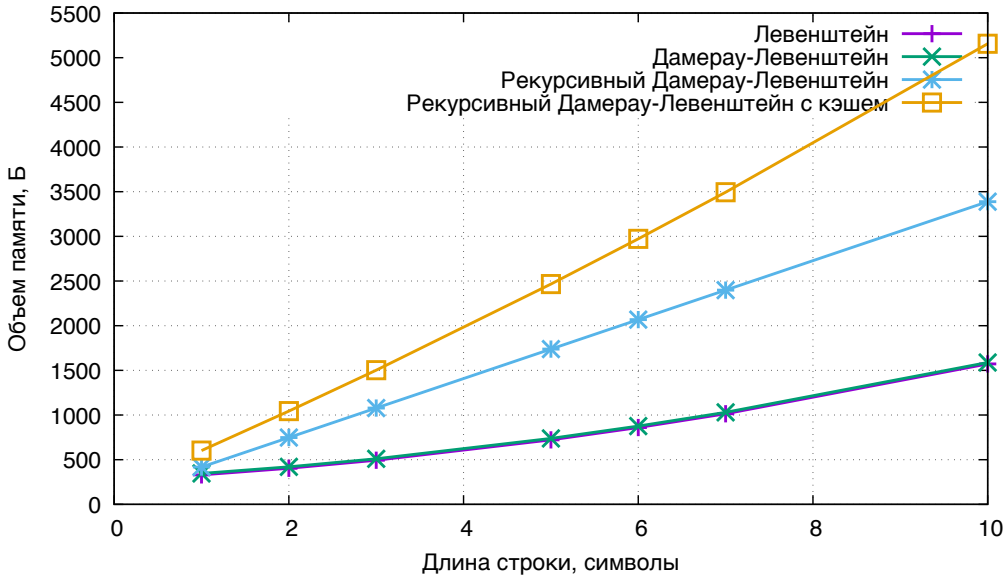

Рисунок 4.4 — Зависимость потребляемой памяти при вычислении редакционного расстояния от длины строк

Заключение

В результате проведения замеров времени выполнения и потребляемой памяти были сформулированы нижеперечисленные выводы.

Если сравнивать алгоритмы поиска редакционного расстояния по времени выполнения, то очевидно, что рекурсивный алгоритм занимает много больше времени, чем реализации, использующие матрицу результатов. Уже при длине строк равной 6 рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна работает примерно в 107 раз дольше, чем аналогичный матричный. При длине строк, равной 10, он работает дольше в 44833 раза.

Если сравнивать две матричные реализации алгоритмов — поиска расстояния Левенштейна и Дамерау – Левенштейна — время работы второго будет превосходить время работы первого, что связано с наличием дополнительных проверок.

Если сравнивать две рекурсивные реализации алгоритма поиска расстояния Дамерау – Левенштейна, время работы алгоритма без кэша будет превосходить время работы алгоритма с кэшем, что связано с необходимостью повторных вычислений при работе в отсутствие матрицы промежуточных результатов.

Если сравнивать реализации алгоритмов по объёму потребляемой памяти, то рекурсивные алгоритмы уступают матричным при любых длинах строк в связи с тем, что при вызове функции выделяется область собственного стека для нее. Так как в рекурсивных методах много вызовов функций, то потребляемая память выходит больше, чем в матричных, в которых стек выделяется на один вызов функции. Так, при длине строк в 10 символов рекурсивная реализация алгоритма поиска расстояния Дамерау – Левенштейна потребляет в 2,1 раза больше байт памяти, чем матричная реализация.

Алгоритм с кэшированием потребляет в среднем в 1,5 раза больше памяти, чем аналогичный алгоритм без кэширования, так как требуется дополнительная память для хранения матрицы.

Потребление памяти матричными реализациями алгоритмов сравнимо, так как разницу составляют только единичные переменные (например, дополнительная переменная с ценой операции транспозиции в алгоритме поиска расстояния Дамерау – Левенштейна на текущем шаге, которой нет в алгоритме поиска расстояния Левенштейна, в связи с отсутствием в нём данной операции).

В ходе выполнения лабораторной работы была достигнута поставленная цель: были получены навыки программирования, тестирования полученного программного продукта и проведения замеров по результатам работы программы на примере решения задачи о редакционном расстоянии.

В процессе выполнения лабораторной работы были также реализованы все поставленные задачи, а именно:

- были изучены расстояния Левенштейна и Дамерау – Левенштейна;
- были разработаны нерекурсивные алгоритмы поиска расстояний Левенштейна и Дамерау – Левенштейна, рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна и модификация последнего с использованием кэша;
- была выполнена программная реализация данных алгоритмов;
- были проведены замеры потребления памяти (в байтах) и времени работы (в нс) для данных алгоритмов;
- была получена графическая зависимость измеренных величин от длины последовательности символов, предоставляемой на вход алгоритмам;
- был проведен сравнительный анализ данных алгоритмов на основе полученных зависимостей.

Список использованных источников

[1] Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау – Левенштейна [Электронный ресурс]. Режим доступа: *https://www.elibrary.ru/item.asp?id=36907767* (дата обращения: 20.09.2022).

[2] Документация по языку программирования *Go* [Электронный ресурс]. Режим доступа: *https://go.dev/doc/* (дата обращения: 20.09.2022).

[3] Документация по пакетам языка программирования *Go* [Электронный ресурс]. Режим доступа: *https://pkg.go.dev* (дата обращения: 20.09.2022).

[4] GoLand: IDE для профессиональной разработки на *Go* [Электронный ресурс]. Режим доступа: *https://www.jetbrains.com/ru-ru/go/* (дата обращения: 20.09.2022).

[5] Техническая спецификация ноутбука *MacBookAir* [Электронный ресурс]. Режим доступа: *https://support.apple.com/kb/SP869* (дата обращения: 20.09.2022).

[6] *AppleM2* [Электронный ресурс]. Режим доступа: *https://www.notebookcheck.net/Apple-M2-Processor-Benchmarks-and-Specs.632312.0.html* (дата обращения: 10.10.2022).

[7] Исходный код *src/testing/benchmark.go* [Электронный ресурс]. Режим доступа: *https://go.dev/src/testing/benchmark.go* (дата обращения: 10.10.2022).

[8] Документация по функции *mach_absolute_time* [Электронный ресурс]. Режим доступа: *https://go.dev/src/testing/benchmark.go* (дата обращения: 10.10.2022).

Приложение А

В ходе выполнения лабораторной работы в соответствии с поставленными задачами было необходимо произвести замеры потребляемой при выполнении функций, реализующих заданные алгоритмы, памяти. В связи с этим был разработан программный модуль *memory* для измерения потребляемой функцией памяти в байтах (замеры реализованы только для функций, представляющих алгоритмы по заданию лабораторной работы).

«Бенчмарки», использованные для измерения затрачиваемого на исполнение функции времени, предоставляют возможность измерить только память, выделяемую на куче, что не соответствует поставленной задаче.

При вызове функции в языке *Go* для неё выделяется область собственного стека, что особенно критично при рекурсивных реализациях. Соответственно, необходимо иметь возможность измерять потребляемую память и на стеке, так как в ином случае не будут получены реалистичные результаты замеров и построить зависимость, отражающую действительное потребление памяти в зависимости от длины строк, будет невозможно.

В листингах 5.1 — 5.3 приведена реализация основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах.

Листинг 5.1 — Листинг основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах (начало)

```
package algorithms

import (
        "unsafe"
)

var MemoryInfo Metrics

type Metrics struct {
        current int
        max     int
}
```

Листинг 5.2 — Листинг основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах (продолжение листинга 5.1)

```
// сброс рассчитанных значений
func (m *Metrics) Reset() {
        m.current = 0
        m.max = 0
}

// добавление значения к общей сумме потребляемой памяти,
// обновление максимума
func (m *Metrics) Add(v int) {
        m.current += v
        if m.current > m.max {
                m.max = m.current
        }
}

// вычитание значения из общей суммы потребляемой памяти
func (m *Metrics) Done(v int) {
        m.current -= v
}

// получение значения макс. потребления памяти за выполнение функции
func (m *Metrics) Max() int64 {
        return int64(m.max)
}

// получение размера типа данных
// пример вызова: sizeOf[int]()
func sizeOf[T any]() int {
        var v T
        return int(unsafe.Sizeof(v))
}
```

Листинг 5.3 — Листинг основных структур и базового функционала модуля *memory* для измерения потребляемой функцией памяти в байтах (окончание листинга 5.2)

```
// получение полного размера среза (заголовок + элементы)
// пример вызова: sizeOfArray[int](10)
func sizeOfArray[T any](n int) int {
        return sizeOf[[]T]() + n*sizeOf[T]()
}
```

В листингах 5.4 – 5.5 приведена реализация одной из функций (в данном случае функции, реализующей алгоритм поиска расстояния Левенштейна), вычисляющих потребление памяти конкретной функцией, модуля *memory*.

Листинг 5.4 — Листинг функции, вычисляющей потребление памяти функцией, реализующей алгоритм поиска расстояния Левенштейна (начало)

```
func memoryLevenshtein(s1, s2 string) int {
        n1 := len(s1)
        n2 := len(s2)

        s01 := sizeOf[string]() + n1*sizeOf[byte]()
        s02 := sizeOf[string]() + n2*sizeOf[byte]()

        n1r := len([]rune(s1))
        n2r := len([]rune(s2))

        sr01 := sizeOfArray[rune](n1r)
        sr02 := sizeOfArray[rune](n2r)

        res := sizeOf[int]()

        loop1 := 2 * sizeOf[int]()

        m := sizeOf[[][]int]() + sizeOfArray[int](n2r+1)*(n1r+1)

        loop2 := 5 * sizeOf[int]()
```

Листинг 5.5 — Листинг функции, вычисляющей потребление памяти функцией, реализующей алгоритм поиска расстояния Левенштейна (окончание листинга 5.4)

```
        minFunc := sizeOfArray[int](3) + sizeOf[int]()*3

        return s01 + s02 + sr01 + sr02 + res + m + loop1 + loop2 + minFunc
}
```

В листинге 5.6 приведен пример использования функций модуля *memory* в реализации алгоритма поиска расстояния Левенштейна.

Листинг 5.6 — Листинг использования функций модуля *memory* в реализации алгоритма поиска расстояния Левенштейна

```
func Levenshtein(s1, s2 string) int {
        MemoryInfo.Reset()
        MemoryInfo.Add(memoryLevenshtein(s1, s2))
        defer MemoryInfo.Done(memoryLevenshtein(s1, s2))
        ...
}
```

Таким образом, данный модуль позволяет измерить полное потребление памяти функциями, реализующими алгоритмы поиска редакционного расстояния, и получить реалистичные данные по результатам измерений.