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 «Анализы Алгоритмов»

на тему: «Динамическое программирование»

Студент группы ИУ7-56Б _____ (Подпись, дата) Чупахин М. Д. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Волкова Л. Л. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

В данной лабораторной работе предстоит исследовать понятие расстояния Левенштейна [1]. Это метрика, которая показывает наименьшее количество операций (вставка, удаление, замена), требуемых для превращения одной строки в другую. Эта метрика играет важную роль в определении сходства между двумя строками.

Интерес к расстоянию Левенштейна был впервые вызван советским математиком Владимиром Левенштейном в 1965 году, когда он исследовал последовательности символов «0» и «1». Позже эта задача была обобщена для произвольного алфавита и получила его имя.

Расстояние Левенштейна имеет широкое применение в теории информации и компьютерной лингвистике. Его используют для решения следующих задач:

— исправление опечаток в словах (в поисковых системах, базах данных, при вводе текста и при автоматическом распознавании отсканированных текстов или речи);

— сравнение текстовых файлов с помощью утилиты diff;

— сравнение геномов, хромосом и белков в области биоинформатики.

Главной целью этой лабораторной работы является описание и исследование алгоритмов, связанных с расстоянием Левенштейна и Дамерау — Левенштейна. Ниже представлены задачи, которые необходимо выполнить для достижения этой цели.

1) Подробно описать алгоритмы для вычисления расстояний Левенштейна и Дамерау — Левенштейна.

2) Разработать программное обеспечение, реализующее следующие алгоритмы:

   — итеративный алгоритм для вычисления расстояния Левенштейна;

   — итеративный алгоритм для вычисления расстояния Дамерау — Левенштейна;

— рекурсивный алгоритм для вычисления расстояния Дамерау — Левенштейна;

— рекурсивный алгоритм с использованием кэширования для вычисления расстояния Дамерау — Левенштейна.

3) Выбрать инструменты для измерения процессорного времени выполнения реализаций алгоритмов.

4) Провести анализ затрат времени и памяти для различных реализаций алгоритмов и выявить факторы, влияющие на эти затраты.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна, также известное как редакционное расстояние, представляет собой метрику, которая позволяет измерить различие между двумя последовательностями символов. Оно определяется как минимальное количество операций редактирования, таких как вставка (I), замена (R) и удаление (D), необходимых для превращения одной строки в другую. Каждая из этих операций имеет свою стоимость:

1) $w(a, b)$ - стоимость замены символа a на символ b;

2) $w(\lambda, b)$ - стоимость вставки символа b;

3) $w(a, \lambda)$ - стоимость удаления символа a.

В данной работе мы будем рассматривать стандартную стоимость, где каждая из этих операций имеет стоимость 1:

— $w(a, b) = 1$, если $a \neq b$, иначе замена не требуется;

— $w(\lambda, b) = 1$;

— $w(a, \lambda) = 1$.

Для обозначения совпадения символов используется символ M (match) с нулевой стоимостью: $w(a, a) = 0$.

Мы также вводим функцию $D(i, j)$, которая представляет собой расстояние Левенштейна между подстроками $S_1[1...i]$ и $S_2[1...j]$.

Расстояние Левенштейна между двумя строками S_1 и S_2 длиной M и N соответственно может быть вычислено с использованием рекуррентной формулы:

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0 \\ i, & \text{j} = 0, \text{i} > 0 \\ j, & \text{i} = 0, \text{j} > 0 \\ \min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \end{cases} & \text{i} > 0, \text{j} > 0 \end{cases} \tag{1.1}$$

Где функция $m(a,b)$ определена как:

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

1.2 Нерекурсивный алгоритм для расстояния Левенштейна

Рекурсивная реализация алгоритма Левенштейна может быть неэффективной по времени при больших значениях M и N, из-за повторных вычислений. Для оптимизации можно использовать итеративную реализацию с использованием матрицы размером $(N+1) \times (M+1)$ для хранения промежуточных значений $D(i,j)$. Значение в ячейке $[i,j]$ будет представлять собой значение $D(S1[1...i], S2[1...j])$. Начальная ячейка заполняется нулем, а затем остальные значения заполняются согласно формуле (1.1).

Однако стоит отметить, что матричный алгоритм может быть малоэффективным по памяти, особенно при больших значениях M и N, так как множество промежуточных значений $D(i,j)$ должно быть хранено в памяти. Для оптимизации по памяти можно использовать рекурсивный алгоритм с кешем, который хранит значения $D(i,j)$, вычисленные на предыдущей итерации, и значения, вычисленные на текущей итерации.

1.3 Расстояние Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна, названное в честь ученых Фредерика Дамерау и Владимира Левенштейна, является мерой разницы между двумя строками символов. Оно определяется как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для превращения одной строки в другую. Это является модификацией расстояния Левенштейна, в которое добавляется операция транспозиции T (transposition).

Расстояние Дамерау — Левенштейна можно вычислить с использованием рекуррентной формулы:

$$D(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0, \\ i, & \text{j} = 0, \text{i} > 0, \\ j, & \text{i} = 0, \text{j} > 0, \\ \min \begin{cases} D(i,j-1)+1, & \\ D(i-1,j)+1, & \text{если i} > 1, \text{j} > 1, \\ D(i-1,j-1)+ & S_1[i] = S_2[j-1], \\ +m(S_1[i],S_2[j]), & S_1[i-1] = S_2[j], \\ D(i-2,j-2)+1, & \end{cases} & \\ \min \begin{cases} D(i,j-1)+1, & \\ D(i-1,j)+1, & \text{иначе.} \\ D(i-1,j-1)+ & \\ +m(S_1[i],S_2[j]), & \end{cases} & \end{cases} \tag{1.3}$$

1.4 Рекурсивный алгоритм для расстояния Дамерау — Левенштейна

Рекурсивный алгоритм для расстояния Дамерау — Левенштейна реализует формулу (1.3) и позволяет вычислить расстояние между двумя строками. Он обладает следующими характеристиками:

1) Для перевода из пустой строки в пустую строку требуется ноль операций.

2) Для перевода из пустой строки в строку a требуется $|a|$ операций.

3) Для перевода из строки a в пустую строку также требуется $|a|$ операций.

4) Для перевода из строки a в строку b требуется выполнить некоторое количество операций вставки, удаления, замены и транспозиции. Порядок выполнения операций не имеет значения.

Минимальная стоимость преобразования достигается путем выбора наименьшей из перечисленных выше операций.

1.5 Рекурсивный алгоритм для расстояния Дамерау — Левенштейна с кэшированием

Рекурсивная реализация алгоритма Дамерау — Левенштейна может быть неэффективной по времени, особенно при больших значениях M и N, из-за повторных вычислений значений расстояний между подстроками. Для оптимизации можно использовать кэш, который хранит значения $D(i, j)$, вычисленные на предыдущей итерации, а также значения, вычисленные на текущей итерации.

Вывод

В данном разделе мы рассмотрели алгоритмы для вычисления расстояний Левенштейна и Дамерау — Левенштейна. Эти алгоритмы могут быть реализованы как рекурсивно, так и итеративно. Они позволяют измерить различие между двумя строками символов, что может быть полезно во многих задачах, таких как автокоррекция и поиск похожих строк.

2 Конструкторская часть

В данном разделе представлены схемы алгоритмов для вычисления расстояний Левенштейна и Дамерау — Левенштейна.

2.1 Разработка алгоритмов

Алгоритмы получают на вход две строки S_1 и S_2.

На рисунке 2.1 представлена схема алгоритма для вычисления расстояния Левенштейна. Алгоритм находит оптимальное расстояние между двумя строками, используя динамическое программирование.

На рисунках 2.2 – 2.5 представлены схемы алгоритмов для вычисления расстояния Дамерау — Левенштейна.

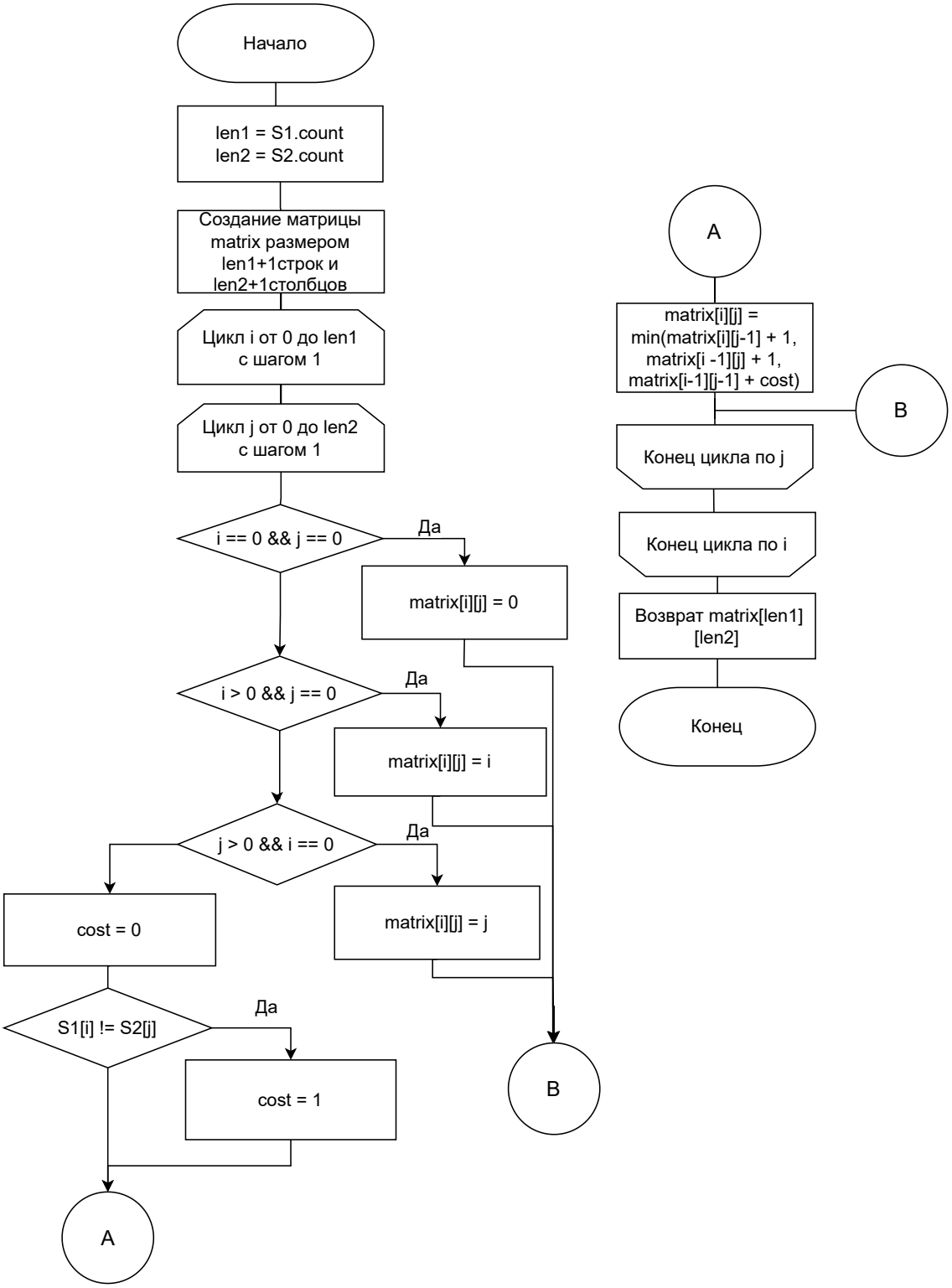

Рисунок 2.1 – Схема алгоритма для вычисления расстояния Левенштейна

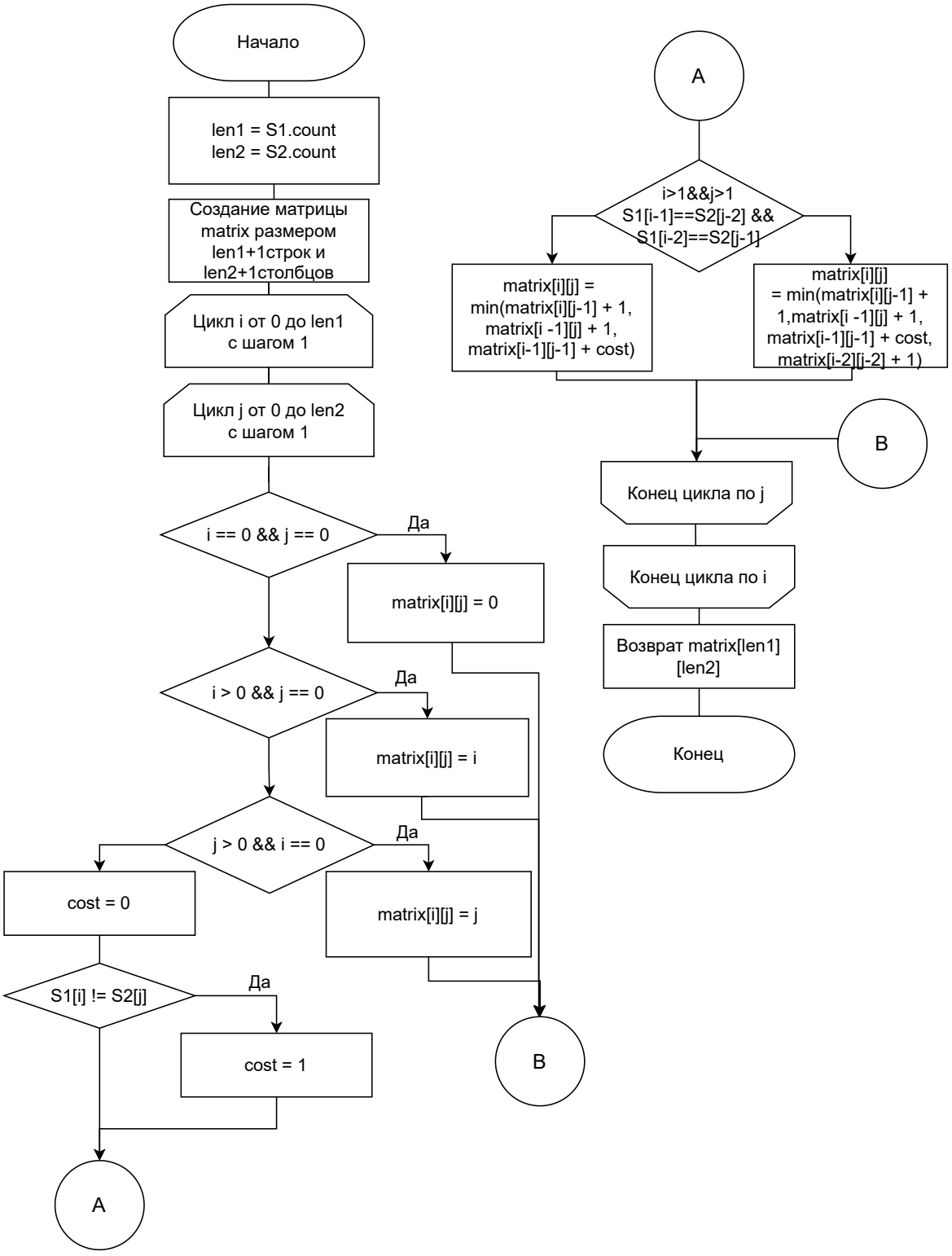

Рисунок 2.2 – Схема алгоритма для вычисления расстояния Дамерау — Левенштейна

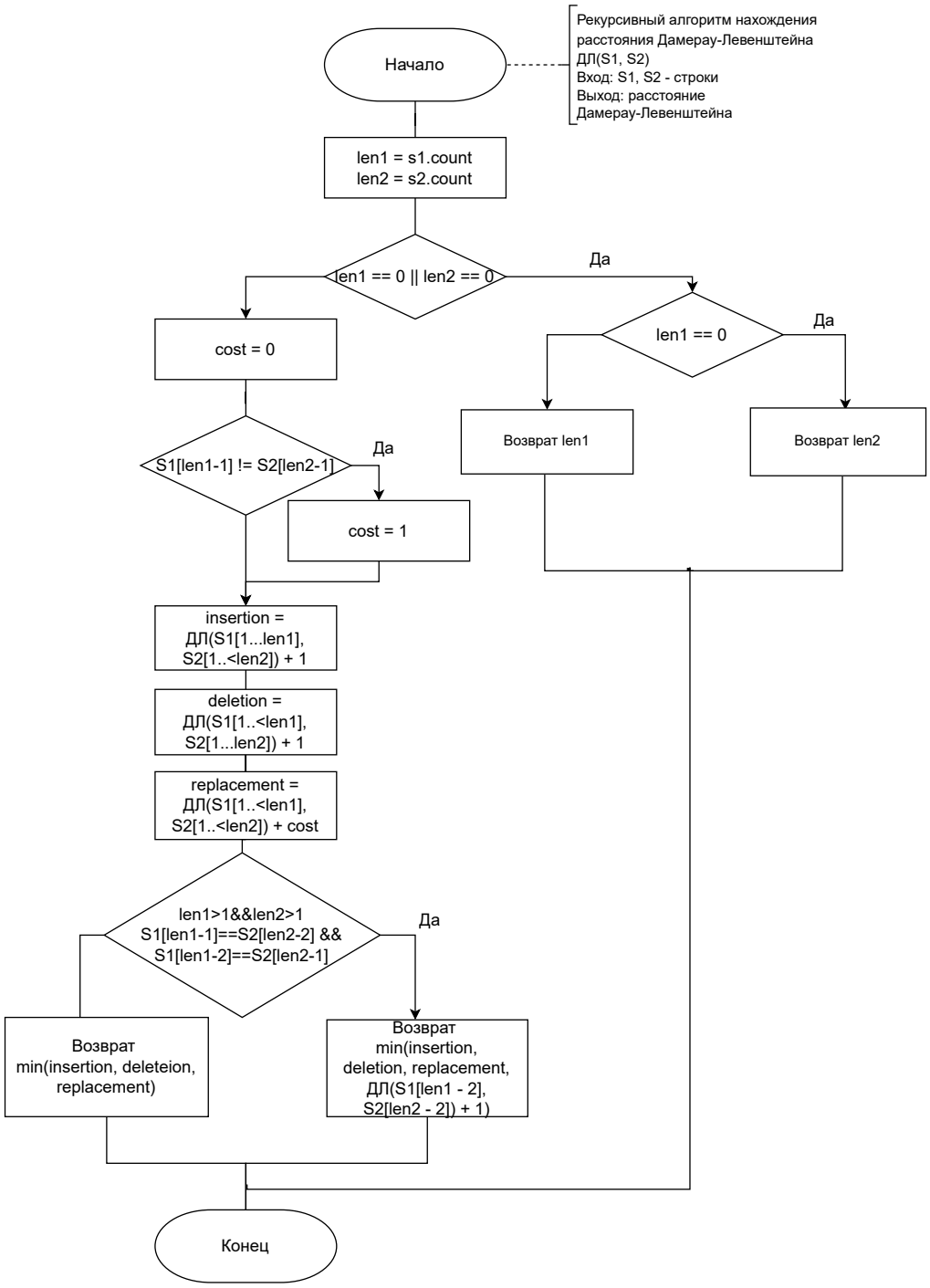

Рисунок 2.3 – Схема рекурсивного алгоритма для вычисления расстояния Дамерау — Левенштейна

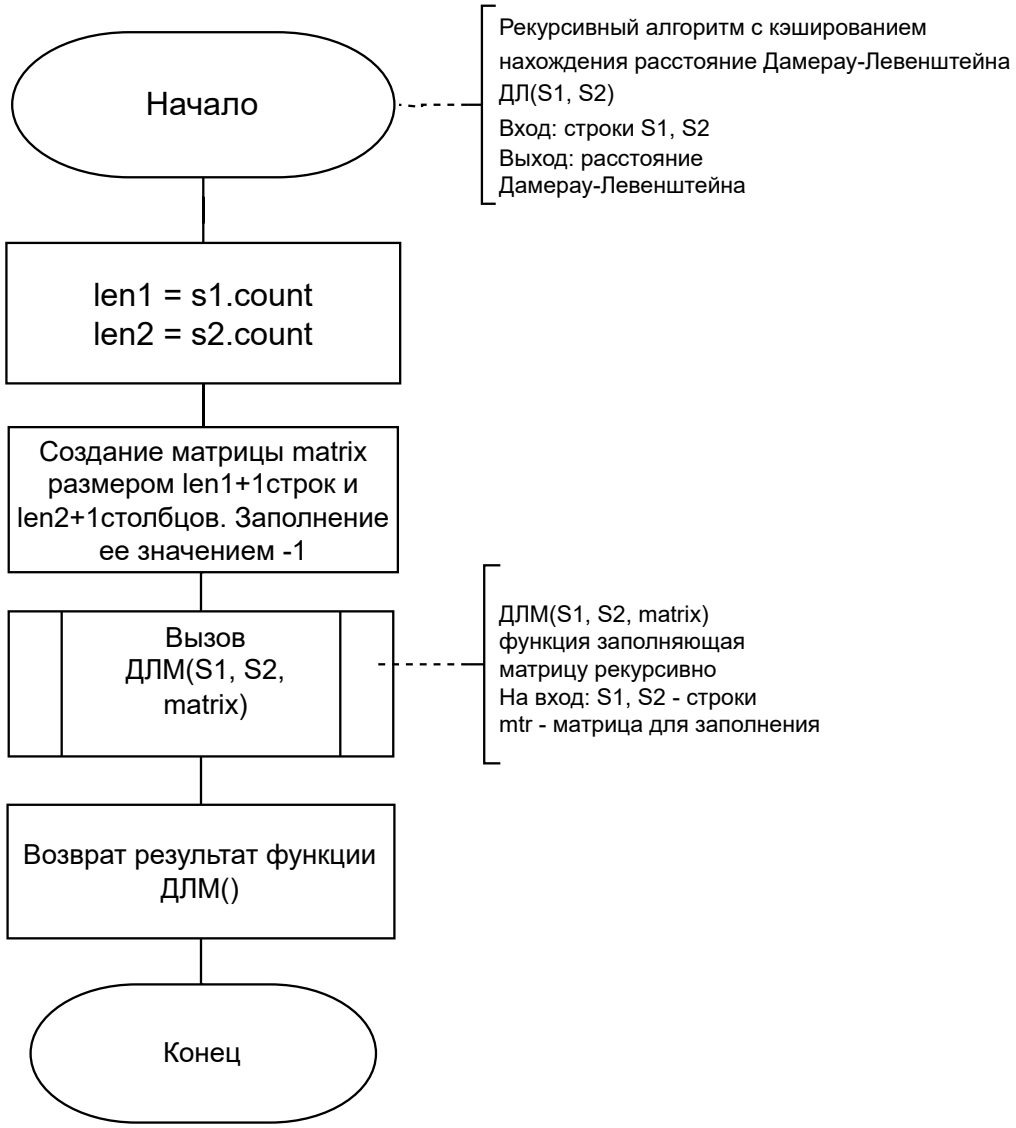

Рисунок 2.4 – Схема рекурсивного алгоритма для вычисления расстояния Дамерау — Левенштейна с использованием кэша

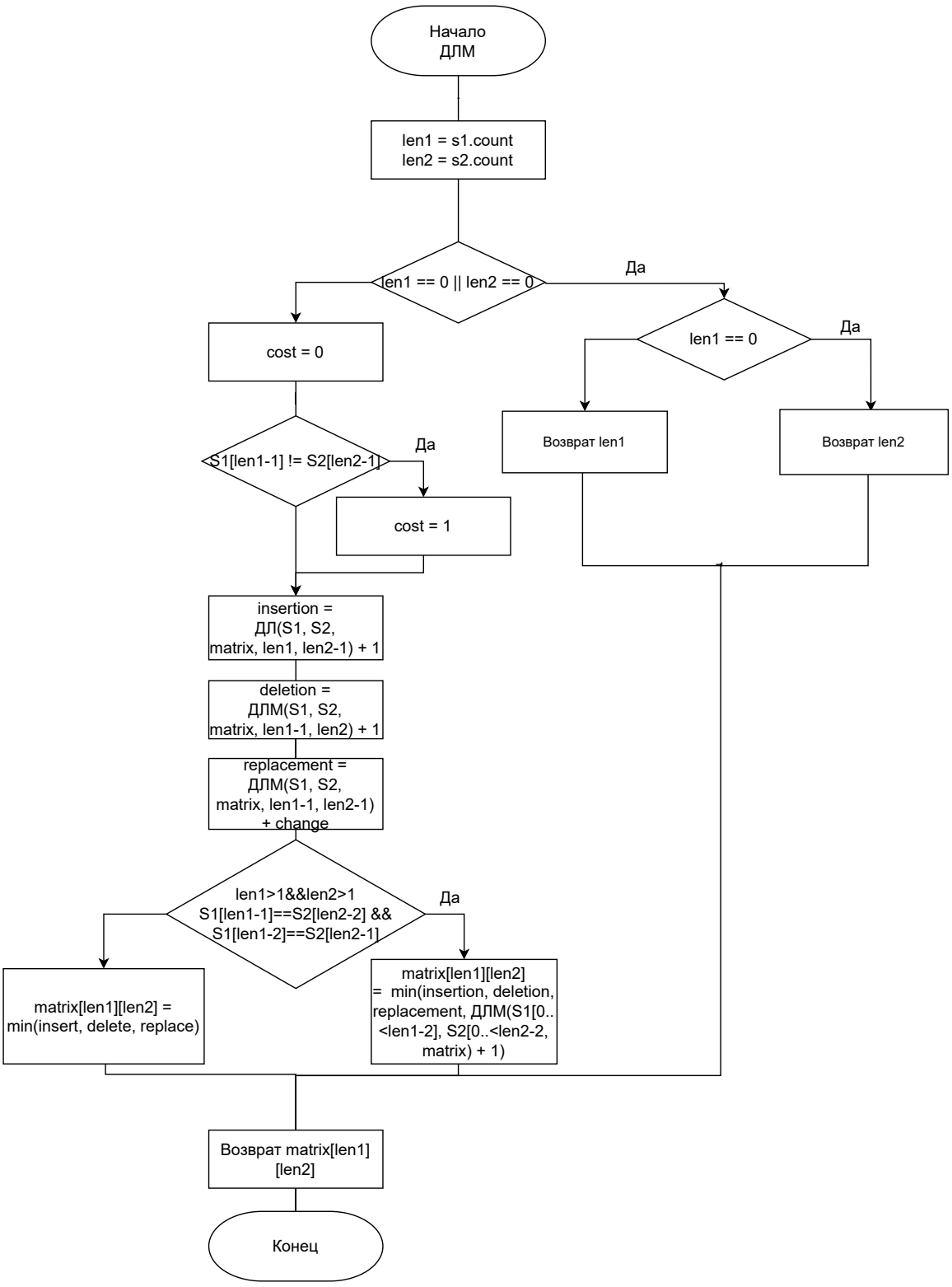

Рисунок 2.5 – Схема алгоритма для рекурсивного заполнения матрицы для вычисления расстояния Дамерау — Левенштейна

Вывод

В данном разделе были представлены схемы алгоритмов для вычисления расстояний Левенштейна и Дамерау — Левенштейна.

3 Технологическая часть

В данной главе представлены требования к программному обеспечению, описаны средства реализации, приведены листинги кода и функциональные тесты.

3.1 Требования к программному обеспечению

Программное обеспечение должно удовлетворять следующим функциональным требованиям: на входе - две строки, на выходе - результаты вычислений всех алгоритмов поиска расстояний, выраженные в виде целых чисел.

Программное обеспечение также должно соответствовать следующим требованиям:

— наличие пользовательского интерфейса для выбора действий;

— способность обрабатывать строки, содержащие символы как латинского, так и кириллического алфавитов;

— предоставление функционала для измерения времени выполнения алгоритмов Левенштейна и Дамерау — Левенштейна.

3.2 Средства реализации

Для разработки данной лабораторной работы был выбран язык программирования Swift [2]. Этот выбор обусловлен возможностью измерения процессорного времени [3], а также возможностью хранить строки, содержащие как кириллические, так и латинские символы [4]. Это удовлетворяет требованиям, предъявляемым к программному обеспечению.

Измерение времени выполнения алгоритмов производится с использованием функции *clock_gettime()* [3].

3.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

— строка — это массив символов типа *char* длиной, равной длине строки;

— длина строки — это целое число типа *int*;

— матрица — это двумерный массив значений типа *int*.

3.4 Сведения о модулях программы

Программа разбита на следующие модули:

— `main.swift` — точка входа в программу, где происходит вызов алгоритмов через интерфейс;

— `Algorithms.swift` — содержит реализации алгоритмов поиска расстояния Левенштейна (только итеративный) и Дамерау — Левенштейна (итеративный, рекурсивный и рекурсивный с кешированием);

— `CPUTimeMeasure.swift` — измеряет время работы алгоритмов с учетом заданного количества повторений;

— `GraphRenderer.swift` — Строит графики для каждого из алгоритмов с учетом заданного количества повторений для каждого алгоритма;

3.5 Реализация алгоритмов

В листингах 3.1 – 3.6 приведены реализации алгоритмов поиска расстояний Левенштейна (только нерекурсивный алгоритм) и Дамерау — Левенштейна (нерекурсивный, рекурсивный и рекурсивный с кешированием).

В листинге 3.7 приведена реализация вывода матрицы.

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы

```
1  private static func nonRecursiveLevenshteinDistance(_ s1:
       String, _ s2: String, print isPrinted: Bool = false) -> Int {
2      let len1 = s1.count
3      let len2 = s2.count
4
5      guard len1 > 0 else { return len2 }
6      guard len2 > 0 else { return len1 }
7
8      let s1Array = Array(s1)
9      let s2Array = Array(s2)
10
11     var matrix = Array(repeating: Array(repeating: 0, count:
           len2 + 1), count: len1 + 1)
12
13     for i in 0...len1 {
14         matrix[i][0] = i
15     }
16     for j in 0...len2 {
17         matrix[0][j] = j
18     }
19
20     for i in 1...len1 {
21         for j in 1...len2 {
22             let cost = s1Array[i - 1] == s2Array[j - 1] ? 0 : 1
23             matrix[i][j] = min(
24                 matrix[i - 1][j] + 1,          // Deletion
25                 matrix[i][j - 1] + 1,          // Insertion
26                 matrix[i - 1][j - 1] + cost    // Replacement
27             )
28         }
29     }
30     if isPrinted {
31         print(Constants.nonRecursiveLevenshteinMatrix)
32         outputMatrix(matrix: matrix, s1: s1, s2: s2)
33     }
34
35     return matrix[len1][len2]
36 }
```

Листинг 3.2 – Функция нахождения расстояния Дамерау — Левенштейна с использованием матрицы

```
1  private static func nonRecursiveDamerauLevenshteinDistance(_
       s1: String, _ s2: String, print isPrinted: Bool = false) ->
       Int {
2      let len1 = s1.count
3      let len2 = s2.count
4
5      guard len1 > 0 else { return len2 }
6      guard len2 > 0 else { return len1 }
7
8      let s1Array = Array(s1)
9      let s2Array = Array(s2)
10
11     var matrix = Array(repeating: Array(repeating: 0, count:
           len2 + 1), count: len1 + 1)
12
13     for i in 0...len1 {
14         matrix[i][0] = i
15     }
16
17     for j in 0...len2 {
18         matrix[0][j] = j
19     }
20
21     for i in 1...len1 {
22         for j in 1...len2 {
23             let cost = s1Array[i - 1] == s2Array[j - 1] ? 0 : 1
24
25             matrix[i][j] = min(
26                 matrix[i - 1][j] + 1,          // Deletion
27                 matrix[i][j - 1] + 1,          // Insertion
28                 matrix[i - 1][j - 1] + cost    // Replacement
29             )
30             if i > 1 && j > 1 && s1Array[i - 1] == s2Array[j -
                   2] && s1Array[i - 2] == s2Array[j - 1] {
31                 matrix[i][j] = min(matrix[i][j], matrix[i -
                       2][j - 2] + cost)  // Transposition
32             }
33         }
34     }
```

Листинг 3.3 – Продолжение листинга 3.2

```
    if isPrinted {
        print(Constants.nonRecursiveDamerauLevenshteinMatrix)
        outputMatrix(matrix: matrix, s1: s1, s2: s2)
    }

    return matrix[len1][len2]
}
```

Листинг 3.4 – Функция нахождения расстояния Дамерау — Левенштейна рекурсивно

```
1  private static func recursiveDamerauLevenshteinDistance(_ s1:
       [Character], _ s2: [Character], _ len1: Int, _ len2: Int) ->
       Int {
2      guard len1 > 0 else { return len2 }
3      guard len2 > 0 else { return len1 }
4
5      let cost = s1[len1 - 1] == s2[len2 - 1] ? 0 : 1
6
7      let deletion = recursiveDamerauLevenshteinDistance(s1, s2,
         len1 - 1, len2) + 1
8      let insertion = recursiveDamerauLevenshteinDistance(s1, s2,
         len1, len2 - 1) + 1
9      let replacement = recursiveDamerauLevenshteinDistance(s1,
         s2, len1 - 1, len2 - 1) + cost
10
11     var transposition = Int.max
12
13     if len1 > 1 && len2 > 1 && s1[len1 - 1] == s2[len2 - 2] &&
         s1[len1 - 2] == s2[len2 - 1] {
14       transposition = recursiveDamerauLevenshteinDistance(s1,
           s2, len1 - 2, len2 - 2) + cost
15     }
16
17     return min(deletion, insertion, replacement, transposition)
18 }
```

Листинг 3.5 – Функция нахождения расстояния Дамерау — Левенштейна рекурсивно с кешированием

```
1  private static func recursiveCacheDamerauLevenshteinDistance(_
       s1: [Character], _ s2: [Character], _ len1: Int, _ len2:
       Int, _ memo: inout [[Int]]) -> Int {
2
3      guard len1 > 0 else {
4          memo[len1][len2] = len2
5          return len2
6      }
7      guard len2 > 0 else {
8          memo[len1][len2] = len1
9          return len1
10     }
11
12     guard memo[len1][len2] == -1 else { return memo[len1][len2]
           }
13
14     let cost = s1[len1 - 1] == s2[len2 - 1] ? 0 : 1
15
16     let deletion = recursiveCacheDamerauLevenshteinDistance(s1,
           s2, len1 - 1, len2, &memo) + 1
17     let insertion =
           recursiveCacheDamerauLevenshteinDistance(s1, s2, len1,
           len2 - 1, &memo) + 1
18     let substitution =
           recursiveCacheDamerauLevenshteinDistance(s1, s2, len1 -
           1, len2 - 1, &memo) + cost
19
20     var transposition = Int.max
21
22     if len1 > 1 && len2 > 1 && s1[len1 - 1] == s2[len2 - 2] &&
           s1[len1 - 2] == s2[len2 - 1] {
23         transposition =
               recursiveCacheDamerauLevenshteinDistance(s1, s2,
               len1 - 2, len2 - 2, &memo) + cost
24     }
25
26     memo[len1][len2] = min(deletion, insertion, substitution,
           transposition)
```

Листинг 3.6 – Продолжение листинга 3.5

```
    return memo[len1][len2]
}

private static func recursiveCacheDamerauLevenshteinDistance(_ s1: String, _ s2: String, print isPrinted: Bool = false) -> Int {
    let len1 = s1.count
    let len2 = s2.count
    let s1Array: [Character] = Array(s1)
    let s2Array: [Character] = Array(s2)
    var memo = Array(repeating: Array(repeating: -1, count: len2 + 1), count: len1 + 1)

    let distance = recursiveCacheDamerauLevenshteinDistance(s1Array, s2Array, s1.count, s2.count, &memo)
    if isPrinted {
        print(Constants.recursiveCacheDamerauLevenshteinMatrix)
        outputMatrix(matrix: memo, s1: s1, s2: s2) }

    return distance
}
```

Листинг 3.7 – Функции вывода матрицы для алгоритмов поиска расстояния Левенштейна и Дамерау — Левенштейна

```
private static func outputMatrix(matrix: [[Int]], s1: String,
    s2: String) {
    let len1 = s1.count
    let len2 = s2.count

    let s1 = Array(s1)
    let s2 = Array(s2)

    print("   |   |", terminator: "")
    for char in s2 {
        print(" \(char) |", terminator: "")
    }
    print()

    print("---|---|", terminator: "")
    for _ in s2 {
        print("---|", terminator: "")
    }
    print()

    for i in 0...len1 {
        if i == 0 {
            print("   |", terminator: "")
        } else {
            print(" \(s1[i - 1]) |", terminator: "")
        }

        for j in 0...len2 {
            print(String(format: "%3d|", matrix[i][j]),
                terminator: "")
        }
        print()
    }
    print()
}
```

3.6 Функциональные тесты

В таблице приведены функциональные тесты для алгоритмов вычисления расстояний Левенштейна и Дамерау — Левенштейна. Все тесты были успешно пройдены.

Таблица 3.1 – Дополнительные функциональные тесты

Входные данные		**Расстояние и алгоритм**			
Строка 1	**Строка 2**	**Левенштейна**	**Дамерау — Левенштейна**		
		Итеративный	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешом**
книга	книги	1	1	1	1
кот	кошка	3	3	3	3
дом	дача	3	3	3	3
птица	рыба	6	6	6	6
привет	превет	1	1	1	1
кот	кто	2	1	1	1

Вывод

В данной главе были представлены требования к программному обеспечению, описаны средства реализации, приведены листинги кода алгоритмов и функциональные тесты, подтверждающие корректность работы алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени:

— Процессор: Apple M1 Pro [5]

— Оперативная память: 32 ГБайт.

— Операционная система: macOS Ventura 13.5.2. [6]

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На изображении 4.1 представлена иллюстрация работы разработанного программного продукта. Конкретно, демонстрируются результаты выполнения алгоритмов для вычисления расстояний Левенштейна и Дамерау — Левенштейна на примере двух строк: "кот"и "скат". Кроме того, на рисунке показываются матрицы, используемые алгоритмами, которые требуют промежуточные матрицы для выполнения расчетов.

```
Menu:
1. Calculate Levenshtein distance
2. Measure CPU time
3. Draw graphs
0. Exit

Choose Menu option: 1
Enter the first string:
кот
Enter the second string:
скат
Calculating Levenshtein distances:
Non Recursive Damerau-Levenshtein Distance Algorithm Matrix:
   |   | с | к | а | т |
---|---|---|---|---|---|
   |  0|  1|  2|  3|  4|
 к |  1|  1|  1|  2|  3|
 о |  2|  2|  2|  2|  3|
 т |  3|  3|  3|  3|  2|

Recursive Damerau-Levenshtein Distance Algorithm with cache matrix:
   |   | с | к | а | т |
---|---|---|---|---|---|
   |  0|  1|  2|  3|  4|
 к |  1|  1|  1|  2|  3|
 о |  2|  2|  2|  2|  3|
 т |  3|  3|  3|  3|  2|

Non Recursive Levenshtein Distance Algorithm Matrix:
   |   | с | к | а | т |
---|---|---|---|---|---|
   |  0|  1|  2|  3|  4|
 к |  1|  1|  1|  2|  3|
 о |  2|  2|  2|  2|  3|
 т |  3|  3|  3|  3|  2|

Non Recursive Damerau-Levenshtein Distance Algorithm   : 2
Recursive Damerau-Levenshtein Distance Algorithm       : 2
Recursive Cache Damerau-Levenshtein Distance Algorithm : 2
Non recursive Levenshtein Distance Algorithm           : 2
```

Рисунок 4.1 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау — Левенштейна

4.3 Временные характеристики

Результаты эксперимента замеров по времени приведены в таблице. Замеры проводились на одинаковых длин строк от 1 до 10 с шагом 1.

Таблица 4.1 – Замер по времени для строк, размер которых от 1 до 10

Длина (символ)	**Время, мкс**			
	Левенштейн	**Дамерау-Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кеша	**С кешом**
1	3.1361	3.8568	1.898	2.3434
2	5.3403	6.4971	9.519	8.1462
3	8.1427	9.384	47.9058	8.1427
4	11.2672	13.8797	252.1432	35.5997
5	15.5059	18.0829	1333.726	60.5605
6	19.8499	22.8499	7295.0763	95.598
7	26.445	31.291	41053.4	143.19
8	31.605	39.135	219978.084	196.504
9	37.632	53.435	1222987.568	259.023
10	46.456	60.68	6645267.384	336.657

Сравнение производится на основе данных, представленных в таблице 4.1. Данные для всех алгоритмов представлены на рисунке 4.2. Данные для всех алгоритмов, не включая рекурсивный Дамерау — Левенштейна представлены на рисунке 4.3

Из-за того, что у алгоритма поиска расстояния Дамерау — Левенштейна задействуется дополнительная операция, которая замедляет алгоритм - он оказывается медленнее алгоритма поиска расстояния Левенштейна. Так же видно, что рекурсивный алгоритм очень быстро становится крайне не эффективным и непригодным к использованию.

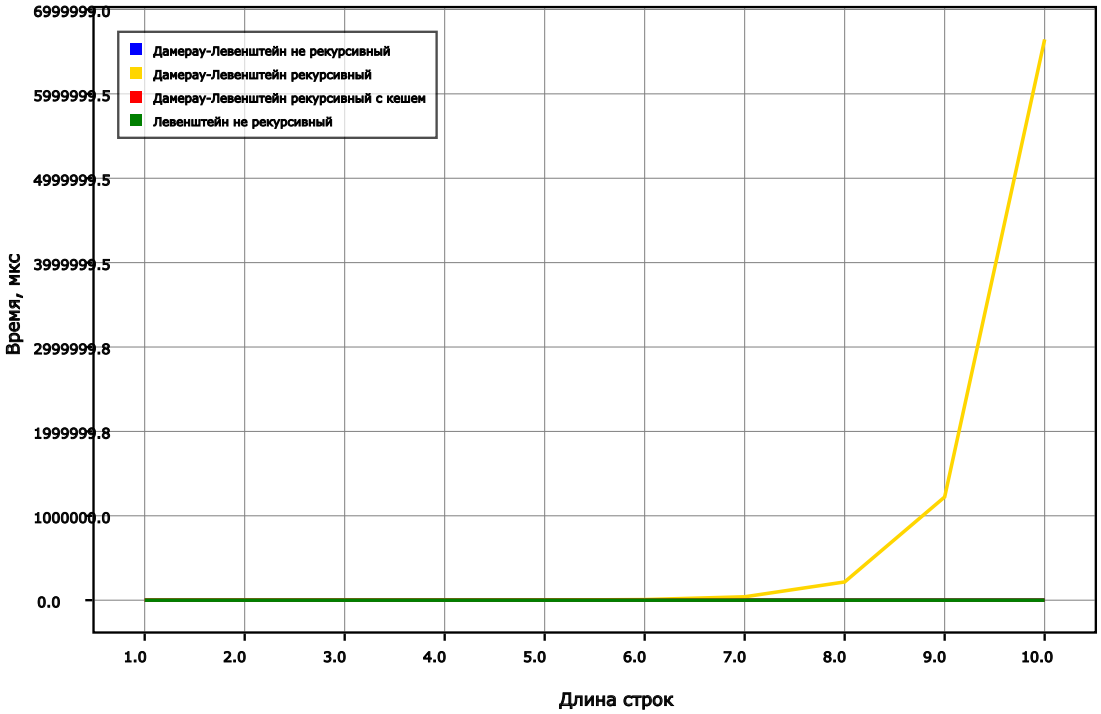

Рисунок 4.2 – Сравнение по времени реализаций алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна

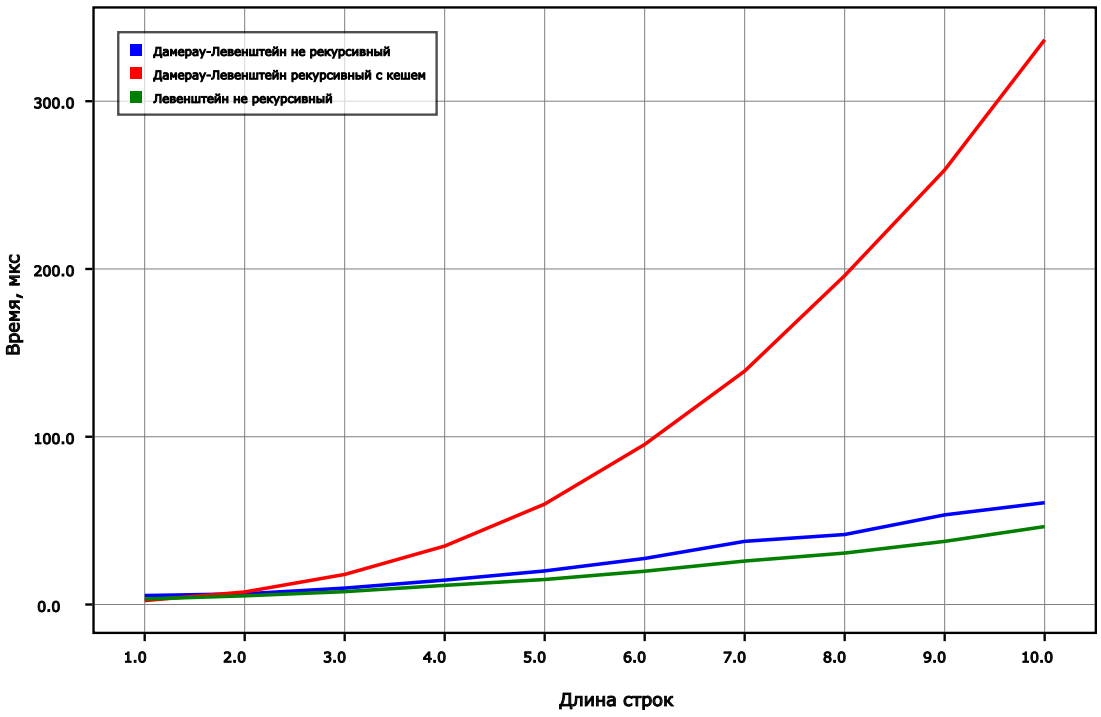

Рисунок 4.3 – Сравнение по времени реализаций алгоритмов без рекурсивного Дамерау — Левенштейна

4.4 Характеристики по памяти

Алгоритмы для вычисления расстояния Левенштейна и Дамерау–Левенштейна не различаются по потреблению памяти. Максимальная глубина стека вызовов при использовании рекурсивной реализации равна сумме длин входных строк. Следовательно, максимальное использование памяти оценивается следующим образом:

$$(size(str1) + size(str2)) \cdot (8 \cdot size(Int) + size(Character)) \tag{4.1}$$

где $size$ - оператор, вычисляющий размер объекта, $str1$ и $str2$ - строки, $Character$ - тип данных для символов строки, и Int - целочисленный тип.

В случае итеративной реализации потребление памяти теоретически можно оценить следующим образом:

$$\begin{aligned}(size(str1) + 1) \cdot (size(str2) + 1) \cdot size(Int) + 3 \cdot size(Int) + \\ + (n + m) \cdot size(Character).\end{aligned} \tag{4.2}$$

4.5 Вывод

В данном разделе проведено сравнение затрат времени и памяти между алгоритмами вычисления расстояния Левенштейна и Дамерау – Левенштейна. Наименее времязатратным оказался итеративный алгоритм для расчета расстояния Левенштейна.

Из собранных данных видно, что рекурсивная реализация алгоритма неэффективна на длинах строк, больших 5. Поэтому рекомендуется использовать рекурсивные алгоритмы только для строк малых размерностей (содержащих 1-4 символа).

Помимо этого, важно отметить, что алгоритм вычисления расстояния Дамерау — Левенштейна предпочтителен в случаях, когда возможны ошибки в транспозиции символов при печати текста, даже несмотря на его

некоторое увеличение затрат по времени и памяти по сравнению с алгоритмом Левенштейна.

Рекурсивная реализация алгоритма для расчета расстояния Дамерау — Левенштейна требует больше времени по сравнению с итеративной реализацией.

Заключение

Результаты данного исследования позволяют сделать несколько важных выводов. В частности, было установлено, что время выполнения алгоритмов вычисления расстояний Левенштейна и Дамерау — Левенштейна возрастает в геометрической прогрессии при увеличении длины строк. Среди всех реализаций алгоритмов наилучшие показатели по времени демонстрируют матричная реализация алгоритма Дамерау — Левенштейна и его рекурсивная версия с использованием кеша.

Основной целью данной лабораторной работы было изучение и описание особенностей задач динамического программирования, а именно алгоритмов Левенштейна и Дамерау — Левенштейна. Ниже представлены выполненные задачи.

1) Проведено подробное описание алгоритмов для вычисления расстояния Левенштейна и Дамерау — Левенштейна.

2) Разработано программное обеспечение, реализующее следующие алгоритмы:

   — Нерекурсивный метод вычисления расстояния Левенштейна.

   — Нерекурсивный метод вычисления расстояния Дамерау — Левенштейна.

   — Рекурсивный метод вычисления расстояния Дамерау — Левенштейна.

   — Рекурсивный метод вычисления расстояния Дамерау — Левенштейна с использованием кеша.

3) Выбраны инструменты для измерения процессорного времени выполнения реализаций алгоритмов.

4) Проведен анализ временных и памятных затрат программы, чтобы выявить влияющие на них характеристики и факторы.

Цели и задачи исследования были успешно достигнуты, и полученные результаты позволяют лучше понять и оценить эффективность различных методов вычисления расстояний Левенштейна и Дамерау — Левенштейна.

Список использованных источников

1 И. Левенштейн В. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: Издательство «Наука», Доклады АН СССР, 1965. Т. 163.

2 Документация к Swift [Электронный ресурс]. — Режим доступа: `https://developer.apple.com/documentation/swift` (дата обращения: 12.09.2023).

3 Измерение процессорного времени [Электронный ресурс]. — Режим доступа: `https://man7.org/linux/man-pages/man3/clock_gettime.3.html` (дата обращения: 14.09.2023).

4 Структура String [Электронный ресурс]. — Режим доступа: `https://developer.apple.com/documentation/swift/string` (дата обращения: 15.09.2023).

5 Процессоры M1 [Электронный ресурс]. — Режим доступа: `https://developer.apple.com/documentation/apple-silicon` (дата обращения: 15.09.2023).

6 Операционная система macOS. [Электронный ресурс]. — Режим доступа: `https://developer.apple.com/macos/` (дата обращения: 15.09.2023).