|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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

**Лабораторная работа №1**

**Расстояние Левенштейна**

|  |  |
| --- | --- |
| Студент: | Уласик Е.А. |
| Группа: | ИУ7-51 |
| Преподаватель: | Волкова Л.Л. |

*2019 г.*

**Оглавление**

[**Введение** 3](#_Toc21709518)

[**Задачи работы** 4](#_Toc21709519)

[**1.** **Аналитическая часть** 5](#_Toc21709520)

[**1.1.** **Необходимые определения** 5](#_Toc21709521)

[**1.2.** **Расстояние Левенштейна** 5](#_Toc21709522)

[**1.3.** **Расстояние Дамерау-Левенштейна** 6](#_Toc21709523)

[**2.** **Конструкторская часть** 8](#_Toc21709524)

[**2.1.** **Рекурсивный алгоритм нахождения расстояние Левенштейна** 8](#_Toc21709525)

[**2.2.** **Матричный алгоритм нахождения расстояния Левенштейна** 9](#_Toc21709526)

[**2.3.** **Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна** 9](#_Toc21709527)

[**2.4.** **Матричный алгоритм нахождения расстояния Дамерау-Левенштейна** 12](#_Toc21709528)

[**2.5.** **Измерение памяти** 14](#_Toc21709529)

[**3.** **Технологическая часть** 15](#_Toc21709530)

[**3.1.** **Рекурсивный алгоритм нахождения расстояние Левенштейна** 15](#_Toc21709531)

[**3.2.** **Матричный алгоритм нахождения расстояния Левенштейна** 15](#_Toc21709532)

[**3.3.** **Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна** 16](#_Toc21709533)

[**3.4.** **Матричный алгоритм нахождения расстояния Дамерау-Левенштейна** 16](#_Toc21709534)

[**4.** **Экспериментальная часть** 18](#_Toc21709535)

[**4.1.** **Тесты** 18](#_Toc21709536)

[**4.2.** **Замеры времени** 18](#_Toc21709537)

[**Заключение** 20](#_Toc21709538)

[**Список литературы** 21](#_Toc21709539)

# **Введение**

Целю лабораторной работы №1 является изучение понятия редакционного расстояния и алгоритмов для его нахождения.

Впервые задачу поставил в 1965 году советский математик Владимир Иосифович Левенштейн при изучении последовательностей 0 – 1 в своем докладе «Двоичные коды с исправлением выпадений, вставок и замещений символов» [1], а впоследствии более общую задачу для произвольного алфавита связали с его именем. Большой вклад в изучение вопроса внёс Дэн Гасфилд в своей работе «Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология» [2].

Расстояние Левенштейна и его обобщения активно применяются для исправления ошибок в слове (в поисковых системах, при вводе текста, при автоматическом распознавании отсканированного текста или речи); в биоинформатике для сравнения генов, хромосом и белков.

Расстояние Дамерау-Левенштейна также используется как мера схожести двух строк относительно редакторских операций. Алгоритм его поиска находит применение в реализации нечёткого поиска, а также в биоинформатике, хотя и изначально алгоритм разрабатывался для сравнения текстов, набранных человеком, т.к. большая часть ошибок при наборе связаны с перестановкой соседних символов. Поэтому метрика Дамерау-Левенштейна часто используется в программах для проверки правописания и при обработке поисковых запросов.

**Задачи работы**

Задачами данной работы являются:

1. Изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения редакционного расстояния между строками.
2. Реализация вышеупомянутых алгоритмов в матричной и рекурсивной формах.
3. Сравнительный анализ матричной и рекурсивной реализаций выбранного алгоритма по таким ресурсам как память и время.
4. Экспериментальное подтверждение различий в эффективности по времени рекурсивной и матричной реализация при помощи разработанного программного обеспечения с функцией замера процессорного времени.
5. Описание и обоснование полученных результатов в отчете о выполненной работе.
6. **Аналитическая часть**

В данном разделе будут даны определения для расстояния Левенштейна и расстояния Дамерау-Левенштейна, а также будет дано формальное описание алгоритмов для поиска расстояний.

* 1. **Необходимые определения**

**Определение 1.** Расстоянием Левенштейна (также функцией Левенштейна) в теории информатики и компьютерной лингвистики называют меру разницы двух последовательностей символов (строк) относительно минимального количества операций вставки одного символа, удаления одного символа и замены одного символа, необходимых для перевода одной последовательности в другую.

**Определение 2.** Расстоянием Дамерау-Левенштейна в теории информатики и компьютерной лингвистики называют меру разницы двух последовательностей символов (строк) относительно минимального количества операций вставки одного символа, удаления одного символа, замены одного символа и транспозиции (перестановки двух соседних символов), необходимых для перевода одной последовательности в другую. Является модификацией расстояния Левенштейна.

**Определение 3.** Редакторскими будем называть операции вставки 1 символа, удаления одного символа и замены одного символа. В расстоянии Дамерау-Левенштейна появляется еще одна редакторская операция – транспозиция соседних символов.

* 1. **Расстояние Левенштейна**

Рассмотрим тривиальные случаи, из которых будет получена формула для рекуррентного вычисления расстояния Левенштейна.

1. Для перевода из пустой строки в пустую требуется нуль операций.
2. Для перевода из строки, состоящей из одного символа, в пустую требуется одна операция (удаление).
3. Для перевода из строки S1 длиной L1 в пустую требуется L1 операций (удаление).
4. Для перевода из пустой строки в строку, состоящую из одного символа, требуется одна операция (вставка).
5. Для перевода из пустой строки в строку S2 длиной L2 требуется L2 операций (вставка).

Следовательно, для перевода из строки S1 в строку S2 требуется выполнить последовательно некоторое количество редакторских операций. Тогда количество операций, необходимых для преобразования строки S1 в строку S2 можно выразить как:

А) Общее количество операций, необходимых для преобразования строки S1 без последнего символа в строку S2, + одна операция удаления, необходимая для удаления последней буквы из строки S1.

Б) Общее количество операций, необходимых для преобразования строки S1 в строку S2 без последнего символа, + одна операция вставки, необходимая для вставки последней буквы в строку S2.

В) Общее количество операций, необходимых для преобразования строки S1 без последнего символа в строку S2 без последнего символа, + одна операция замены, если строки оканчиваются на разные символы.

Г) Общее количество операций, необходимых для преобразования строки S1 без последнего символа в строку S2 без последнего символа, + нуль, если строки оканчиваются на одинаковые символы.

Таким образом, редакторским расстоянием между двумя строками будет минимальный набор из вышеописанных вариантов.

Расстояние Левенштейна между двумя строками 𝑎 и 𝑏 может быть вычислено по формуле 𝐷(|𝑎|, |𝑏|) (здесь и далее запись |𝑎| означает длину строки 𝑎; запись 𝑎[𝑖] означает 𝑖-ый символ строки 𝑎, нумерация символов начинается с единицы), где функция 𝐷(𝑖, 𝑗) определена так, как показано в Формуле 1.

|  |  |
| --- | --- |
|  | (1) |

где функция 𝑚(𝑎, 𝑏) равна нулю, если 𝑎 = 𝑏 и единице в противном случае. Рекурсивный алгоритм напрямую реализует эту формулу.

Можно также реализовать этот алгоритм в матричном виде. Для двух строк S1 и S1 с длинами L1 и L2 соответственном потребуется матрица D размерами L1+1 на L2+1. В ячейке матрица D[i][j] будет храниться результат функции

𝐷(i, j). Расстояние Левенштейна будет находиться в ячейке матрицы D[L1][L2].

* 1. **Расстояние Дамерау-Левенштейна**

Расстояние Дамера-Левенштейна является модификацией расстояние Левенштейна. При поиске данного расстояния в список редакторских операций вносится еще одна операция – транспозиция.

Расстояние Дамерау-Левенштейна может быть найдено так, как показано в Формуле 2.

|  |  |
| --- | --- |
|  | (2) |

где функция 1(𝑎𝑖 ≠ 𝑏𝑗) равна единице, если 𝑎𝑖 ≠ 𝑏𝑖 и нулю в противном случае. Функция *Ma,b(i,j)* определена так, как показано в Формуле 3.

|  |  |
| --- | --- |
|  | (3) |

Формула выводится аналогично формуле для нахождения расстояния Левенштейна.

Также можно ввести матричную версию алгоритма. Которая представляет из себя матричную версию алгоритма для нахождения расстояния Левенштейна, но добавляется проверка для транспозиции.

1. **Конструкторская часть**

В данном разделе будут даны схемы алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна в матричной и рекурсивной версиях.

* 1. **Рекурсивный алгоритм нахождения расстояние Левенштейна**

Рекурсивная версия алгоритма нахождения расстояния Левенштейна, показанный на Рисунке 1, напрямую реализует Формулу (1): выполняется 3 вызова и выбирается минимальный из результатов этих вызовов.



Рисунок 1 — Схема рекурсивного алгоритма нахождения расстояния Левенштейна.

* 1. **Матричный алгоритм нахождения расстояния Левенштейна**

Матричная версия алгоритма нахождения расстояния Левенштейна, показанная на Рисунке 2, реализована с использованием вложенного цикла для заполнения матрицы с редакционными расстояниями.



Рисунок 2 — Схема матричного алгоритма нахождения расстояния Левенштейна.

* 1. **Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна**

Рекурсивная версия алгоритма нахождения расстояния Дамерау-Левенштейна, показанная на Рисунке 3, реализует Формулу 3. В отличие от расстояния Левенштейна в данном алгоритме существует проверка на возможность совершения транспозиции и в зависимости от нее выполняется разное количество вызовов, среди которых находится минимальное значение.



Рисунок 3 — Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

* 1. **Матричный алгоритм нахождения расстояния Дамерау-Левенштейна**

Матричная версия алгоритма нахождения расстояния Дамерау-Левенштейна, показанная на Рисунке 4, реализована с использованием вложенного цикла для заполнения матрицы с редакционными расстояниями. Единственным исключением является проверка на возможность транспозиции.



Рисунок 4 — Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

* 1. **Измерение памяти**

Рекурсивная реализация работает медленнее относительно матричной реализации, т.к. в ходе ее работы возникают повторные вычисления. При каждом рекурсивном вызове можно передавать либо новую укороченную строку, либо указатель на строку с укороченной длиной. Второй вариант эффективнее по памяти, т.к. не следует выделять память для новых подстрок.

Вызова функции, реализующей матричный способ, происходит один раз и в функцию передаются обе строки сразу. Но в этом алгоритме память будет затрачена на хранение матрицы, а время будет потрачена на заполнение этой матрицы. Однако затраты будут существенно меньше, чем при рекурсивных вызовах.

Пусть имеются строки S1, S2 с длинами len1, len2 соответственно.

В рекурсивных алгоритмах на каждый вызов требуется (len1 + len2\_ \* sizeof(char) байт на хранение строк, 2 \* sizeof(int) байт на хранение длин строк. Для оценки максимальной необходимой памяти необходимо умножить это количество памяти на глубину рекурсивного дерева вызовов. Глубина этого дерева равна len1 + len2.

1. **Технологическая часть**

Для реализации вышеописанных алгоритмов был выбран язык программирования высокого уровня Python версии 3.7.2. Замеры времени работы алгоритма будут выполнены с использованием функции perf\_counter\_ns из библиотеки time. Данная функция возвращает текущее время из счетчика системного монитора в наносекундах, замерив время до и после работы функции, и по разности этих значений узнать сколько времени было потрачено на выполнение данного процесса. В данном разделе будут представлены листинги программы.

* 1. **Рекурсивный алгоритм нахождения расстояние Левенштейна**

В листинге 1 показана реализация рекурсивного алгоритма нахождения расстояния Левенштейна.

Листинг 1. Функция нахождения расстояния Левенштейна рекурсивно

import Foundation  
  
func recursive\_distance(str\_1: String, str\_2: String) -> Int {  
 let len\_1 = str\_1.count  
 let len\_2 = str\_2.count  
  
 if (len\_1 == 1 && len\_2 == 1) {  
 if (str\_1 == str\_2) {  
 return 0  
 } else {  
 return 1  
 }  
 } else if (len\_1 == 0 || len\_2 == 0) {  
 return max(len\_1, len\_2)  
 }  
  
 var d = 1  
 if (String(str\_1.suffix(1)) == String(str\_2.suffix(1))) {  
 d = 0  
 }  
  
 return min(recursive\_distance(str\_1: str\_1, str\_2: String(str\_2.prefix(len\_2 - 1))) + 1,  
 recursive\_distance(str\_1: String(str\_1.prefix(len\_1 - 1)), str\_2: str\_2) + 1,  
 recursive\_distance(str\_1: String(str\_1.prefix(len\_1 - 1)), str\_2: String(str\_2.prefix(len\_2 - 1))) + d)  
}

* 1. **Матричный алгоритм нахождения расстояния Левенштейна**

В листинге 2 показана реализация матричного алгоритма нахождения расстояния Левенштейна.

Листинг 2. Функция нахождения расстояния Левенштейна матрично

import Foundation  
  
func matrix\_distance(str\_1: String, str\_2: String, values: Bool = false) -> Int {  
 let len\_1 = str\_1.count  
 let len\_2 = str\_2.count  
  
 var matrix = [[Int]]()  
 var row\_1 = [Int]()  
 var row\_2 = [Int]()  
 for x in 0..<len\_2 + 1 {  
 row\_1.append(x)  
 }  
 var d : Int  
 for i in 1..<len\_1 + 1 {  
 row\_2 = [i]  
 for j in 1..<len\_2 + 1 {  
 d = 1  
 if (str\_1[str\_1.index(str\_1.startIndex, offsetBy: i - 1)] == str\_2[str\_2.index(str\_2.startIndex, offsetBy: j - 1)]) {  
 d = 0  
 }  
  
 row\_2.append(  
 min(row\_1[j] + 1,  
 row\_2[j - 1] + 1,  
 row\_1[j - 1] + d)  
 )  
 }  
 matrix.append(row\_1)  
 row\_1 = row\_2  
 }  
 matrix.append(row\_1)  
 if (values) {  
 print("Матрица \"Матричного метода\": ")  
 print\_matrixы(matrix: matrix)  
 }  
  
 return row\_1[row\_1.count - 1]  
}

* 1. **Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна**

В листинге 3 показана реализация рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

Листинг 3. Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

import Foundation  
  
func recursive\_distance(str\_1: String, str\_2: String) -> Int {  
 let len\_1 = str\_1.count  
 let len\_2 = str\_2.count  
  
 if (len\_1 == 1 && len\_2 == 1) {  
 if (str\_1 == str\_2) {  
 return 0  
 } else {  
 return 1  
 }  
 } else if (len\_1 == 0 || len\_2 == 0) {  
 return max(len\_1, len\_2)  
 }  
  
 var d = 1  
 if (String(str\_1.suffix(1)) == String(str\_2.suffix(1))) {  
 d = 0  
 }  
  
 if (len\_1 > 1 && len\_2 > 1) {  
 if (str\_1[str\_1.index(str\_1.endIndex, offsetBy: -1)] == str\_2[str\_2.index(str\_2.endIndex, offsetBy: -2)] &&  
 str\_1[str\_1.index(str\_1.endIndex, offsetBy: -2)] == str\_2[str\_2.index(str\_2.endIndex, offsetBy: -1)]) {  
 return min(recursive\_distance(str\_1: str\_1, str\_2: String(str\_2.prefix(len\_2 - 1))) + 1,  
 recursive\_distance(str\_1: String(str\_1.prefix(len\_1 - 1)), str\_2: str\_2) + 1,  
 recursive\_distance(str\_1: String(str\_1.prefix(len\_1 - 1)), str\_2: String(str\_2.prefix(len\_2 - 1))) + d,  
 recursive\_distance(str\_1: String(str\_1.prefix(len\_1 - 2)), str\_2: String(str\_2.prefix(len\_2 - 2))) + 1)  
 }  
 }  
  
 return min(recursive\_distance(str\_1: str\_1, str\_2: String(str\_2.prefix(len\_2 - 1))) + 1,  
 recursive\_distance(str\_1: String(str\_1.prefix(len\_1 - 1)), str\_2: str\_2) + 1,  
 recursive\_distance(str\_1: String(str\_1.prefix(len\_1 - 1)), str\_2: String(str\_2.prefix(len\_2 - 1))) + d)  
}

* 1. **Матричный алгоритм нахождения расстояния Дамерау-Левенштейна**

В листинге 4 показана реализация матричного алгоритма нахождения расстояния Дамерау-Левенштейна.

Листинг 4. Функция нахождения расстояния Дамерау-Левенштейна матрично

import Foundation  
  
func damerau\_matrix\_distance(str\_1: String, str\_2: String, values: Bool = false) -> Int {  
 let len\_1 = str\_1.count + 1  
 let len\_2 = str\_2.count + 1  
  
 var matrix = [[Int]]()  
 for i in 0..<len\_1 {  
 matrix.append(Array(repeating: 0, count: len\_2))  
 }  
  
 for x in 0..<len\_1 {  
 matrix[x][0] = x  
 }  
 for x in 0..<len\_2 {  
 matrix[0][x] = x  
 }  
  
 var d: Int;  
 for i in 1..<len\_1 {  
 for j in 1..<len\_2 {  
 d = 1  
 if (str\_1[str\_1.index(str\_1.startIndex, offsetBy: i - 1)] == str\_2[str\_2.index(str\_2.startIndex, offsetBy: j - 1)]) {  
 d = 0  
 }  
  
 matrix[i][j] = min(matrix[i][j - 1] + 1,  
 matrix[i - 1][j] + 1,  
 matrix[i - 1][j - 1] + d)  
 if (i > 1 && j > 1 &&  
 str\_1[str\_1.index(str\_1.startIndex, offsetBy: i - 1)] == str\_2[str\_2.index(str\_2.startIndex, offsetBy: j - 2)] &&  
 str\_1[str\_1.index(str\_1.startIndex, offsetBy: i - 2)] == str\_2[str\_2.index(str\_2.startIndex, offsetBy: j - 1)]) {  
 matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + 1)  
 }  
 }  
 }  
  
 if (values) {  
 print("Матрица \"метода Дамерау\": ")  
 print\_matrix(matrix: matrix)  
 }  
  
 return matrix[len\_1 - 1][len\_2 - 1]  
}

1. **Экспериментальная часть**

В данном разделе будут показаны результаты тестов и замеры времени.

* 1. **Тесты**

В таблице 1 показаны ожидаемые результаты, которые должны быть получены в результате проверки программы. В первых двух столбцах показаны тестируемы строки, в столбцах три, четыре и пять указан результат алгоритма. В ходе тестирования программы все ожидаемые результаты совпали с реальными.

Таблица 1

Ожидаемые результаты.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Строка 1 | Строка 2 | Расстояние Левенштейна  (матрица) | Расстояние Дамерау-Левенштейна  (рекурсия) | Расстояние Дамерау-Левенштейна  (матрица) |
| (пусто) | (пусто) | 0 | 0 | 0 |
| кошка | мышка | 2 | 2 | 2 |
| лот | пот |  |  |  |
| (пусто) | Крот | 4 | 4 | 4 |
| сеть |  | 4 | 4 | 4 |
| развлечение | увлечения | 4 | 4 | 4 |
| акт | кат | 2 | 1 | 1 |
| мгту | мтгу | 2 | 1 | 1 |
| float | flop | 2 | 2 | 2 |
| for | rof | 2 | 2 | 2 |
| radically | forcefully | 7 | 7 | 7 |
| miscellaneous | exponentially | 13 | 13 | 13 |
| word | work | 1 | 1 | 1 |
| слово | слово | 0 | 0 | 0 |
| строка | собака | 3 | 3 | 3 |
| текст | торт | 3 | 3 | 3 |
| торт | тост | 1 | 1 | 1 |
| kitten | sitting | 3 | 3 | 3 |
| мурлыкание | мурылкание | 2 | 1 | 1 |
| frostbolt | fortsbotl | 5 | 3 | 3 |
| ginger | iggnre | 4 | 3 | 3 |
| north | south | 2 | 2 | 2 |

* 1. **Замеры времени**

На Рисунках 5, 6 показаны графики, полученные в результате выполнения программы. На данных графиках показано усредненное количество времени, затраченное на поиск редакционного расстояния между двумя строками. По оси OX на графиках указаны длины строк, по оси OY – время работы алгоритма.

Рисунок 5. График сравнения рекурсивных реализаций.

Рисунок 6. График сравнения матричных реализаций.

Анализируя эти графики, мы можем понять, что рекурсивная реализация качественно хуже матричной. Расстояния Левенштейна и Дамерау-Левенштейна близки по своим временным показателям.

**Заключение**

В ходе выполнения данной работы были выполнены все поставленные задачи:

1. Были изучены алгоритмы нахождения редакционного расстояния между двумя строками.
2. По изученному материалу были построены схемы алгоритмов, реализующих поиск редакционного расстояния.
3. Дана оценка памяти, которая будет затрачена при работе алгоритмов.
4. По построенным схемам было разработано программное обеспечение с функцией замера времени для проведения исследования и экспериментального подтверждения различий в эффективности алгоритмов.
5. На основе полученных результатов был составлен отчёт с информацией о проделанной работе.

В результате стало понятно, что использование рекурсивных реализаций алгоритмов нахождения редакционного расстояния неэффективно по времени и по объему затраченной памяти относительно матричных реализаций.

# **Список литературы**

1. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845-848.
2. Гасфилд. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, 2003.
3. Hjelmqvist, Sten (26 March 2012), Fast, memory efficient Levenshtein algorithm
4. Boytsov, Leonid (May 2011). "Indexing methods for approximate dictionary searching". Journal of Experimental Algorithmics.