Государственное образовательное учреждение высшего
профессионального образования
“Московский государственный технический университет имени
Н.Э.Баумана”

Дисциплина: Анализ алгоритмов

Лабораторная работа № 1

Поиск редакционного растояния

Гибадулин О.Н.
Студент группы ИУ7-52

2019 г.

Содержание

Введение

Каждый день пользователи компьютеров сталкиваются с вводом текста, поэтому распознование ошибок в нём является актуальной задачей, которая решается использованием алгоритмов, определяющих степень различия двух строк.

Цель работы: изучение методов решения задачи о нахождении расстояния Левенштейна и Дамерау-Левенштейна.

Задачи работы:

1. изучение алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна между двумя строками;
2. сравнение матричной и рекурсивной реализаций алгоритма определения расстояния Дамерау-Левенштейна;
3. экспериментальное подтверждение корректной работы алгоритмов, исследование временных затрат;
4. описание и обоснование полученных результатов.

1 Аналитический раздел

В данном разделе будет дано понятие редакционного расстояния, а также описаны алгоритмы его нахождения.

1.1 Описание алгоритмов

Расстояние Левенштейна (также известное как редакционное расстояние или дистанция редактирования) – это мера различия двух последовательностей символов (строк) относительно минимального количества операций вставки, удаления и замены, необходимых для перевода одной строки в другую. Для одинаковых строк расстояние редактирования равно нулю.[1]

В 1965 году советский математик Владимир Иосифович Левенштейн разработал алгоритм, который позволяет численно оценить, насколько похожа одна строка на другую.[3]

Основная идея алгоритма состоит в том, чтобы посчитать минимальное количество операций удаления, вставки и замены, которые необходимо сделать над одной из строк, чтобы получить вторую. При этом все три операции обладают так называемым штрафом или ценой, равной 1.

Эти алгоритмы активно применяются[1]:

1. в поисковых системах для нахождения объектов или записей по имени;
2. в базах данных при поиске с неполно-заданным или неточно-заданным именем;
3. для исправления ошибок при вводе текста;
4. для исправления ошибок в результате автоматического распознавания отсканированного текста или записей речи;
5. в приложениях, связанных с автоматической обработкой текстов.

1.2 Расстояние Левенштейна

Пусть S1 и S2 – две строки (длиной n и m соответственно) над некоторым алфавитом, тогда редакционное расстояние d(S1,S2) можно подсчитать по следующей рекуррентной формуле:

$$d(S1,S2) = D(n,m) = \begin{cases} 0; i = 0, j = 0 \\ i; j = 0, i > 0 \\ j; i = 0, j > 0 \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+m(S_1[i],S_2[j]), \end{cases} \end{cases}$$

где m(a,b) равна нулю, если a = b и единице в противном случае; min(a,b,c) возвращает наименьший из аргументов.

Рассмотрим формулу более подробно. Шаг по i символизирует удаление из первой строки, шаг по j – вставку в первую строку, а шаг по обоим индексам символизирует отсутствие изменений или замену символа в первой строке на символ во второй. В нетривиальном случае необходимо выбрать минимальную стоймость из трех вариантов. Вставка/удаление будет в любом случае стоить одну операцию, а вот замена может не понадобиться, если символы равны – тогда шаг по обоим индексам бесплатный.

1.3 Расстояние Дамерау — Левенштейна

Если к списку разрешённых операций добавить транспозицию (которая также имеет стоимость 1), получается расстояние Дамерау — Левенштейна. Дамерау показал, что 80 % человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа, и ошибка в символе.[2] При этих ошибках с добавлением транспозиции редационное расстояние находится за меньшее количество шагов.

$$d(S1,S2) = D(n,m) = \begin{cases} 0; i = 0\, j = 0 \\ i; j = 0\, i > 0 \\ j; i = 0\, j > 0 \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+m(S_1[i],S_2[j]) \\ \begin{cases} D(i-2,j-2)+1; i > 1, j > 1, S_1[i] = S_2[j-1], S_1[i-1] = S_2[j] \\ \infty, else \end{cases} \end{cases} \end{cases},$$

1.4 Вывод

В данном разделе было дано определение редакционного расстояния, были описаны алгоритмы нахождения расстояний Левенштейна и Дамерау

— Левенштейна.

2 Конструкторский раздел

В данном разделе будет формализован и описан процесс вычисления редакционного расстояния с помощью диаграммы idef0, а также в соответствии с формулами алгоритмов, приведенными в аналитической части работы, будут рассмотрены схемы алгоритмов нахождения расстояний: Левенштейна (нерекурсивный подход) и Дамерау-Левенштейна (рекурсивный и нерекурсивный подходы).

2.1 Формализация процесса

В данном пункте представлена idef0-диаграммы для описания функциональной модели процесса нахождения редакционного расстояния двух строк (рис. 2.1.1, рис. 2.1.2).

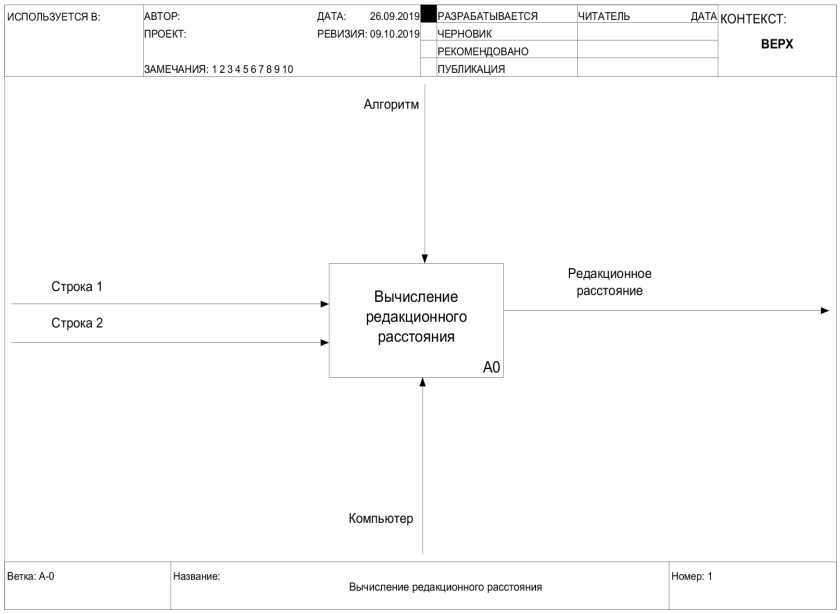

Рисунок 2.1.1. – функциональная схема верхнего уровня процесса нахождения редакционного расстояния

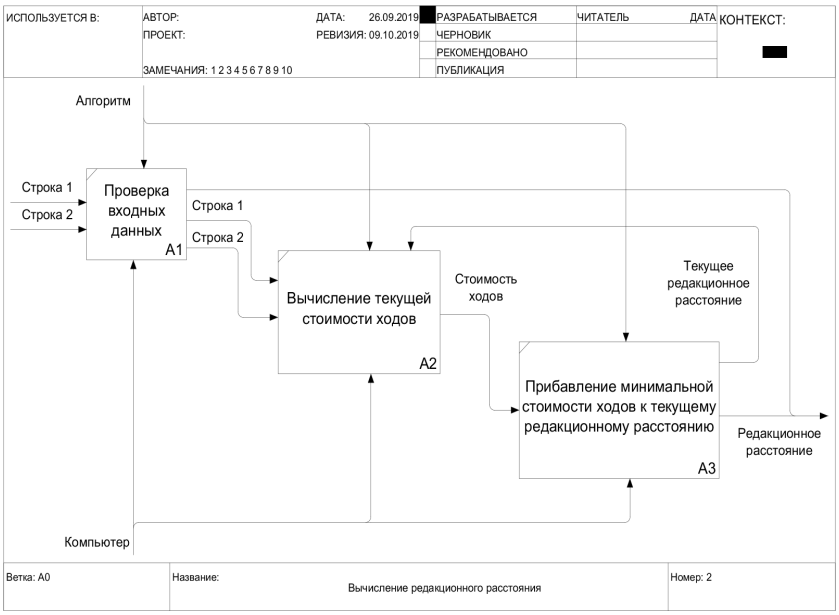

Рисунок 2.1.2. – функциональная схема второго уровня процесса нахождения редакционного расстояния

2.2 Разработка Алгоритмов

В данном пункте представлены схемы алгоритмов нахождения редакционных расстояний. Все алгоритмы начинаются с проверки длины входных строк. Если длина первой строки (n) равна 0, то конечное редакционное растояние равно длине второй строки (m), также аналогично обратное.

В итеративном алгоритме Левенштейна (рис. 2.2.1) учавствуют два массива: первый (A) отвечает за предыдущую строку матрицы преобразования, второй (B) за текущую строку. Для каждой ячейки матрицы, начиная со второй строки, на основе этих массивов вычисляются стоимости операций: добавление, удаление и замена. Результатом, который записывается в текущую ячейку матрицы, является наименьшая по стоимости операция. После прохождения всей матрицы конечным редакционным расстоянием является элемент, находящийся в последней ячейке второго массива (B).

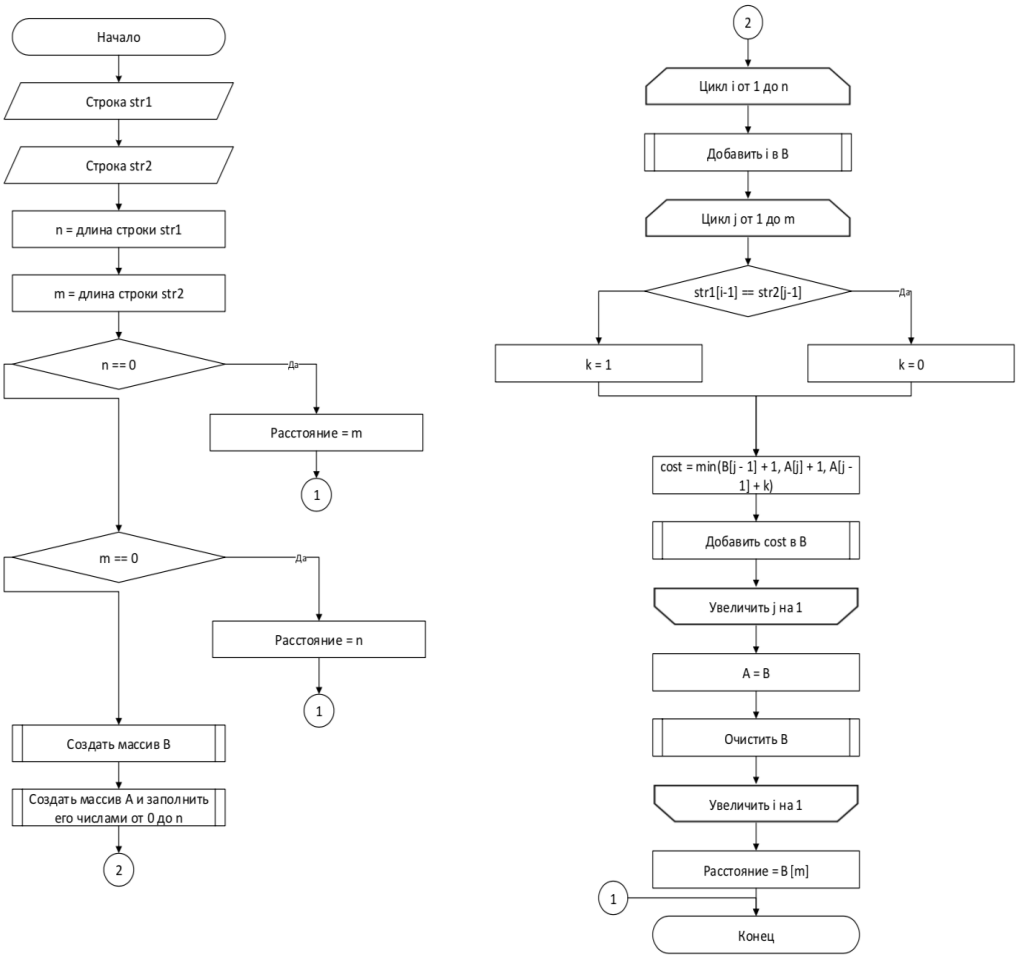

Рисунок 2.2.1. – схема итеративного алгоритма поиска расстояния Левенштейна

В рекурсивной реализации алгоритма Дамерау-Левенштейна (рис. 2.2.2) операций добавления, удаления, замены и транспозиции осуществляются путём уменьшения длины строк в соответствии с их формулами и дальнейшего рекурсивного вызова функции. В качестве конечного редакционного расстояния берется минимальная по стоимости операция.

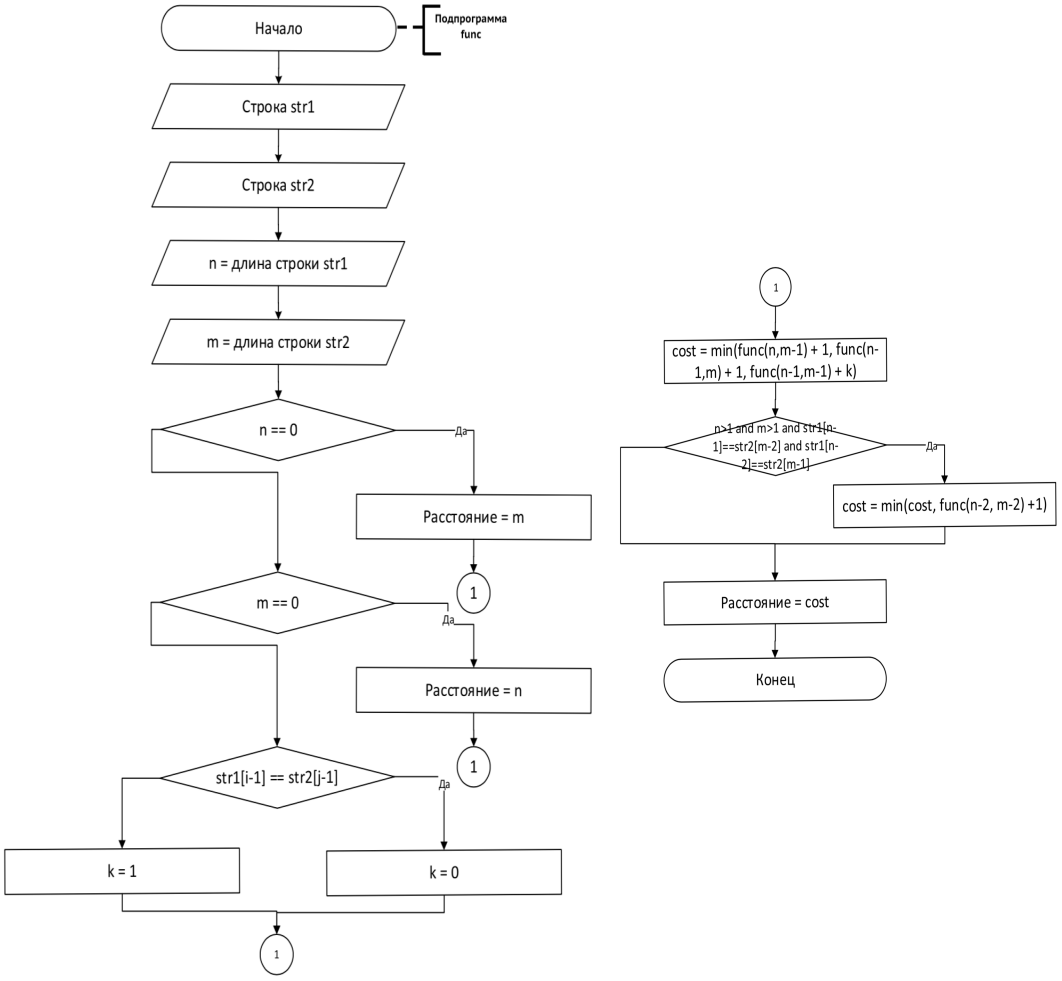

Рисунок 2.2.2. – схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна

Итеративный алгоритм Дамерау-Левенштейна (рис. 2.2.3) аналогичен итеративному алгоритму Левенштейна за исключением того, что на этапе вычисления стоимостей добавляется операция транспозиции, для вычисления которой используется третий массив (C), содержащий элементы пред-предыдущей строки матрицы преобразования.

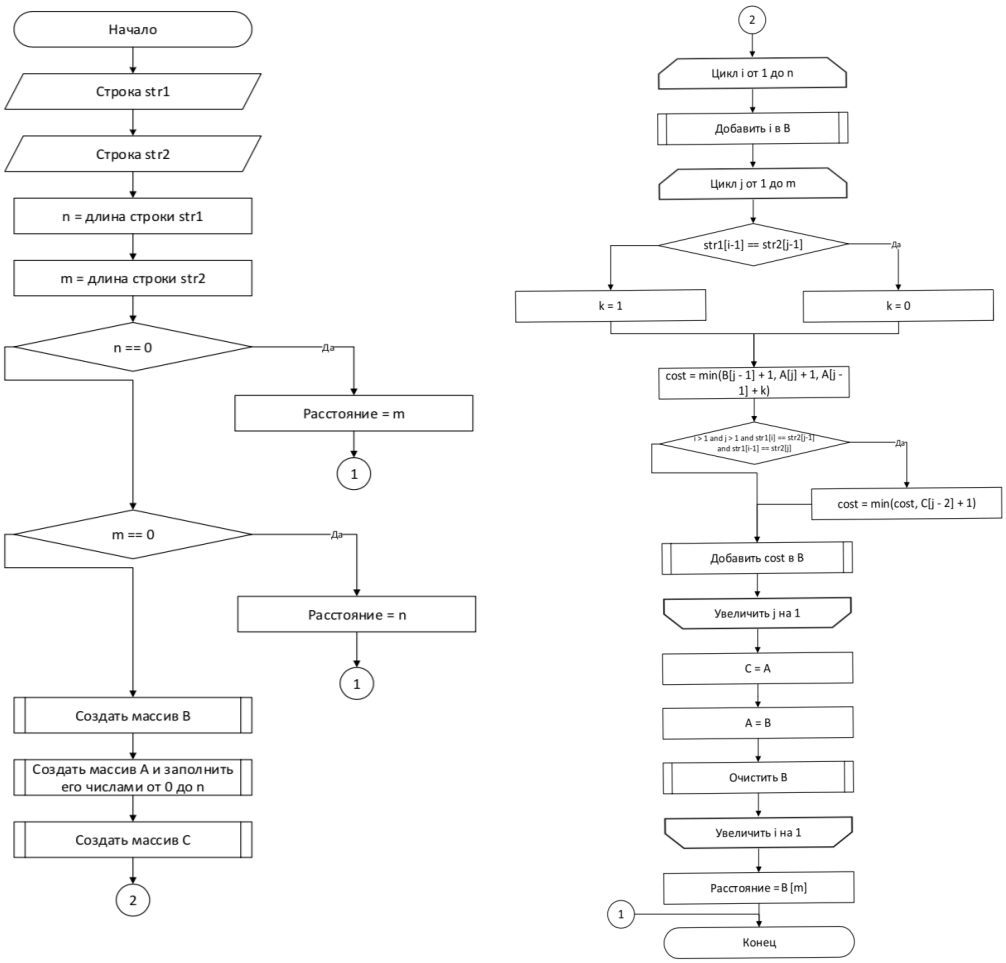

Рисунок 2.2.3. – схема итеративного алгоритма поиска расстояния Дамерау-Левенштейна

2.3 Вывод

В данном разделе был формализован и описан процесс вычисления редакционного расстояния, а также рассмотрены схемы алгоритмов нахождения расстояний: Левенштейна (нерекурсивный подход) и Дамерау-Левенштейна (рекурсивный и нерекурсивный подходы).

3 Технологический раздел

В данном разделе будут рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач, а также представлен листинг кода программы.

3.1 Требования к программному обеспечению

Программное обеспечение должно реализовывать 3 алгоритма нахождения расстояния между двумя строками – итеративный алгоритм Левенштейна и как итеритвный, так и рекурсивный алгоритмы Дамерау-Левенштейна. Пользователь должен иметь возможность произвести замер расстояния для введёной пары строк, а также иметь возможность сравнить время работы этих алгоритмов.

3.2 Средства реализации

Для реализации поставленной задачи был использован язык программирования C++[4]. Проет был выполнен в среде XCode[5]. Для измерения процессорного времени была использована ассемблерная инструкция rdtsc[6].

3.3 Листинг кода

На основе схем алгоритмов, представленных в конструкторском разделе, в соответствии с указанными требованиями к реализации с использованием средств языка C++ было разработано программное обеспечение, содержащее реализации выбранных алгоритмов. В данном пункте приведён листиннг этих реализаций.

Листинг 1. Код нерекурсивной реализации алгоритма Левенштейна

```
size_t levenshteinDistance(const std::string&
    str1, const std::string& str2) {
const size_t len1 = str1.size();
const size_t len2 = str2.size();

if (len1 == 0) {
return len2;
}
if (len2 == 0) {
return len1;
}

std::vector<size_t> prevRow;
for (int i = 0; i < len2 + 1; ++i) {
prevRow.push_back(i);
}
```

```
    std::vector<size_t> curRow;

    for (int i = 1; i < len1 + 1; ++i) {
    curRow.push_back(i);
    for (int j = 1; j < len2 + 1; ++j) {
    const size_t insertionCost = curRow[j - 1] + 1;
    const size_t deletionCost = prevRow[j] + 1;
    const size_t substitutionCost = prevRow[j - 1] +
        match(str1[i - 1], str2[j - 1]);
    const size_t minCost = findMin(insertionCost,
        deletionCost, substitutionCost);
    curRow.push_back(minCost);
    }
    prevRow = curRow;
    curRow.clear();
    }

    return curRow[len2];
    }
```

Листинг 2. Код рекурсивной реализации алгоритма Дамерау - Левенштейна

```
    size_t damerauLevenshteinDistanceRec(const std::
        string& str1, const std::string& str2, const
        size_t len1 , const size_t len2) {
    if (len1 == 0) {
    return len2;
    }
    if (len2 == 0) {
    return len1;
    }

    const size_t insertionCost =
        damerauLevenshteinDistanceRec(str1, str2, len1
        , len2 - 1) + 1;
    const size_t deletionCost =
        damerauLevenshteinDistanceRec(str1, str2, len1
         - 1, len2) + 1;
    size_t substitutionCost =
        damerauLevenshteinDistanceRec(str1, str2, len1
         - 1, len2 - 1) + match(str1[len1 - 1], str2[
        len2 - 1]);

    size_t minCost = findMin(insertionCost,
        deletionCost, substitutionCost);
    if (len1 > 1 &&  len2 > 1 && str1[len1 - 1] ==
        str2[len2 - 2] && str1[len1 - 2] == str2[len2
        - 1]) {
    const size_t transpositionCost =
        damerauLevenshteinDistanceRec(str1, str2, len1
```

```
    - 2, len2 - 2) + 1;
minCost = std::min(minCost, transpositionCost);
}

return minCost;
}
```

Листинг 3. Код нерекурсивной реализации алгоритма Дамерау - Левенштейна

```
size_t damerauLevenshteinDistance(const std::
    string& str1, const std::string& str2) {
const size_t len1 = str1.size();
const size_t len2 = str2.size();

if (len1 == 0) {
return len2;
}
if (len2 == 0) {
return len1;
}

std::vector<size_t> prevPrevRow;
std::vector<size_t> prevRow;
for (int i = 0; i < len1 + 1; ++i) {
prevRow.push_back(i);
}

std::vector<size_t> curRow;

for (int i = 1; i < len1 + 1; ++i) {
curRow.push_back(i);
for (int j = 1; j < len2 + 1; ++j) {
const size_t insertionCost = curRow[j - 1] + 1;
const size_t deletionCost = prevRow[j] + 1;
const size_t substitutionCost = prevRow[j - 1] +
    match(str1[i - 1], str2[j - 1]);

size_t minCost = findMin(insertionCost,
    deletionCost, substitutionCost);
if (i > 1 && j > 1 && str1[i - 1] == str2[j - 2]
   && str1[i - 2] == str2[j - 1]) {
const size_t transpositionCost = prevPrevRow[j -
    2] + 1;
minCost = std::min(minCost, transpositionCost);
}
curRow.push_back(minCost);
}
prevPrevRow = prevRow;
prevRow = curRow;
curRow.clear();
```

```
        }

        return curRow[len2];
        }
```

3.4 Вывод

В данном разделе были рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки, а также был представлен листинг реализаций выбранных алгоритмов.

4 Экспериментальный раздел

В данном разделе будут проведены тесты на корректность работы реализованных алгоритмов, а также проведено исследование временных затрат и затрат по памяти разработанного программного обеспечения, вместе с подробным сравнительным анализом реализованных алгоритмов на основе экспериментальных данных. Все замеры были произведены на процессоре 2,7 GHz Intel Core i5 с паматью 8 ГБ 1867 MHz DDR3.

4.1 Примеры работы

В данном пункте представлены результаты теста на корректность работы алгоритмов при различных входных данных (табл. 4.1.1)

Таблица 4.1.1. Примеры работы алгоритмов

Строка 1	Строка 2	Левенштейн	Дамерау-Левенштейн (рек)	Дамерау-Левенштейн
-	-	0	0	0
-	a	1	1	1
a	-	1	1	1
a	a	0	0	0
a	b	1	1	1
ab	ba	2	1	1
abc	cba	2	2	2
exponential	polynomial	6	6	6

4.2 Сравнительное исследование по времени

Сравнение времени работы алгоритмов Левенштейна и Дамерау-Левенштейна (итеративный и рекусивный) в тактах процессора (табл. 4.2.1, рис. 4.2.1).

Таблица 4.2.1. Сравнение времени работы всех алгоритмов в тактах процессора

Длина строки	Левенштейн	Дамерау-Левенштейн (рек)	Дамерау-Левенштейн
1	173	185	140
2	11253	435	27208
3	23201	1439	15378
4	12680	7955	15352
5	17034	39864	21124
6	26831	241807	268603
7	38716	1324185	89970
8	41160	6632991	47315
9	54295	35725615	97633
10	64736	179362598	76241

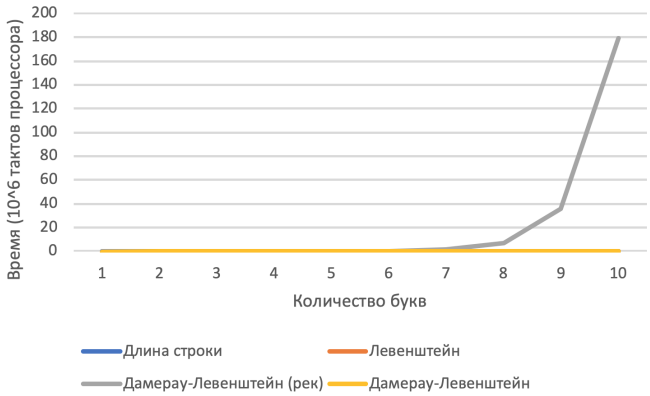

Рисунок 4.2.1. – график зависимости времени работы алгоритмов Левенштейна и Дамерау-Левенштейна (итеративный и рекусивный) от длины входных строк

Сравнение времени работы алгоритмов Левенштейна и Дамерау-Левенштейна (итеративный) в тактах процессора (табл. 4.2.2, рис. 4.2.2).

Таблица 4.2.2. Сравнение времени работы итеративных алгоритмов в тактах процессора

Длина строки	Левенштейн	Дамерау-Левенштейн
100	225	171
200	3680302	4180067
300	12129343	16772758
400	28473496	37478549
500	50684253	66393971
600	77255679	107973959
700	121260856	146656931
800	149217178	201237583
900	208332988	274366249
1000	306863972	347989093

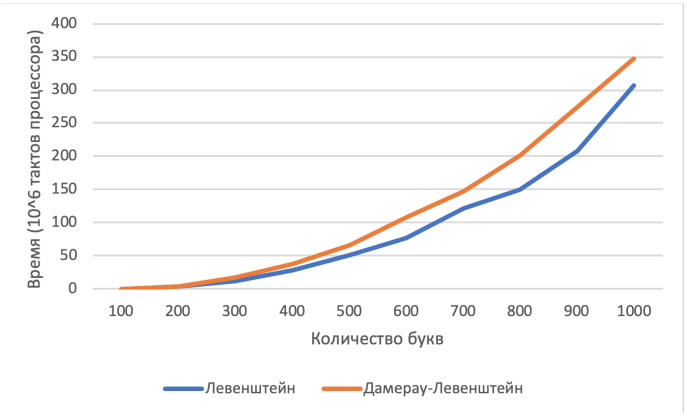

Рисунок 4.2.2. – график зависимости времени работы алгоритмов Левенштейна и Дамерау-Левенштейна (итеративный) от длины входных строк

Итеративные реализации алгоритмов поиска расстояний Дамерау-Левенштейна и Левенштейна имеют схожую ассимптотику.

Время выполнения рекурсивного алгоритма увеличивается экспоненциально, пропорционально количеству рекурсивных вызовов. При сходном немалом размере строк рекурсивный алгоритм сильно проигрывает итеративному. Например при длине строки равной 10 время выполнения итеративной версии алгоритма состовляет 0.04 % от времени рекурсивной версии.

Рекурсивный алгоритм при более простой реализации работает чрезвычайно долго, что делает его использование целесообразным только на малом количестве букв в строках. Итеративный алгоритм значительно превосходит его по эффективности.

4.3 Сравнительное исследование по памяти

Сравнение выделеной памяти при работе алгоритмов (табл. 4.3.3, рис. 4.3.3). В рекурсивной версии алгоритма выделенная память зависит от количества вызовов, поэтому количество выделенной памяти в этом алгоритме увеличивается экспоненциально. В итеративных алгоритмох память выделяется для хранения соседних строк матрицы, длина которых зависит от длины входной строки, поэтому в этих алгоритмах количество выделенной памяти линейно зависит от длины строк.

Таблица 4.3.3. Сравнение выделеной памяти в байтах при работе алгоритмов

Длина строки	Левенштейн	Дамерау-Левенштейн (рек)	Дамерау-Левенштейн
2	68	116	72
4	76	551	80
6	84	2726	88
8	92	13949	96
10	100	73196	104

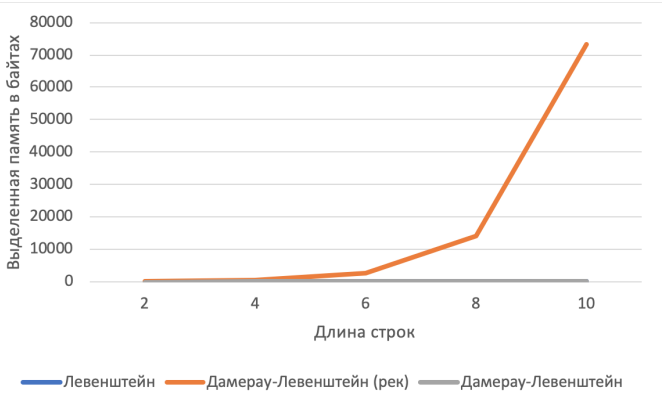

Рисунок 4.3.3. – график зависимости выделенной памяти при работе алгоритмов от длины входных строк

4.4 Вывод

В данном разделе были проведены тесты на корректность работы реализованных алгоритмов, а также проведено исследование временных затрат и затрат по памяти разработанного программного обеспечения, вместе с подробным сравнительным анализом реализованных алгоритмов на основе экспериментальных данных.

Заключение

В ходе выполнения данной лабораторной работы были изучены алгоритмы нахождения редакционного расстояния между двумя строками, было произведено сравнение матричной и рекурсивной реализаций алгоритма, а также проведено исследование временных затрат и затрат по памяти всех алгоритмов.

В аналитическом разделе были описаны алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна. В конструкторском разделе был формализован и описан процесс вычисления редакционного расстояния, разработаны алгоритмы его нахождения. В технологическом разделе были рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач, а также представлен листинг кода программы. В экспериментальном разеде было проведено исследование временных затрат со сравнением матричной и рекурсивной реализаций алгоритма.

Список литературы

[1] В. И. Левенштейн. «Двоичные коды с исправлением выпадений, вставок и замещений символов» - М.: Доклады Академий Наук СССР, 1965.

[2] Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ — 3-е изд. — М.: «Вильямс», 2013. — с. 440.

[3] Д. С. Карахтанов. «Программная реализация алгоритма Левенштейна для устранения опечаток в записях баз данных» [Электронный ресурс] / Молодой ученый. – Режим доступа: https://moluch.ru/archive/19/1966/, свободный. (Дата обращения: 15.09.2019 г.)

[4] ISO/IEC JTC1 SC22 WG21 N 3690 «Programming Languages — C++» [Электронный ресурс]. – Режим доступа: https://devdocs.io/cpp/, свободный. (Дата обращения: 29.09.2019 г.)

[5] Apple «Apple Developer Documentation» [Электронный ресурс]. – Режим доступа: https://developer.apple.com/documentation/, свободный. (Дата обращения: 29.09.2019 г.)

[6] Microsoft «rdtsc» [Электронный ресурс]. – Режим доступа: https://docs.microsoft.com/ru-ru/cpp/intrinsics/rdtsc?view=vs-2019, свободный. (Дата обращения: 29.09.2019 г.)