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

Название: Расстояние Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-55Б (Группа) _____ (Подпись, дата) М.А. Козлов (И.О. Фамилия)

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

Москва, 2020

Содержание

Введение

В данной работе требуется изучить и применить алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, а также получить практические навыки реализации указанных алгоритмов.

Расстояния Левенштейна и Дамерау-Левенштейна применяется:

1) для автозамены, в том числе в поисковых системах;
2) в биоинформатике для сравнения цепочек белков, генов и т.д.

1 Аналитический раздел

1.1 Цель и задачи практики

Цель: реализовать и сравнить по эффективности алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна.

Задачи:

1) Дать математическое описание расстояния Левенштейна и Дамерау-Левенштейна.
2) Разработать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.
3) Реализовать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна.
4) Провести эксперименты по замеру времени работы реализованных алгоритмов.
5) Сравнительный анализ реализованных алгоритмов по затраченному времени и максимально затраченной памяти.

1.2 Алгоритм Левенштейна

Расстояние Левенштейна (или редакционное расстояние) - это минимальное количество редакционных операций, которое необходимо для преобразования одной строки в другую.

Редакционными операциями являются:

1) вставка (I - Insert);
2) удаление (D - Delete);
3) замена (R - Replace);
4) совпадение (M - Match).

Где операции I, D, R имеют штраф 1, а операция M - 0.

Пусть s_1 и s_2 — две строки (длиной M и N соответственно) в некотором алфавите V, тогда расстояние Левенштейна можно подсчитать по рекуррентной формуле 1.1:

$$D(s1[1..i],s2[1..j]) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(s1[1..i],s2[1..j-1] + 1, & \\ D(s1[1..i-1],s2[1..j]) + 1, & j > 0, i > 0 \\ D(s1[1..i-1],s2[1..j-1]) + M(s1[i], s2[j]) & \end{cases} \tag{1.1}$$

Где $s[1..k]$ - подстрока длиной k и $M(a,b) = \begin{cases} 0, & a = b \\ 1, & a \neq b \end{cases}$

В Таблице 1.1 минимальное расстояние между словом "кит"и "скат"равно 2. Последовательность редакторских операций, которая привела к ответу - IMRM.

Таблица 1.1 — Пример работы преобразования слова "кит"в "скат"

	λ	С	К	А	Т
λ	0	1	2	3	4
К	1	1	1	2	3
И	2	2	2	2	3
Т	3	3	3	3	2

1.3 Алгоритм Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, добавлена операция транспозиции (перестановки двух соседних символов) (X - exchange). Операция транспозиции возможна, если символы попарно совпадают.

Пусть s_1 и s_2 — две строки (длиной M и N соответственно) в некотором алфавите V, тогда расстояние Левенштейна можно подсчитать по рекуррентной формуле 1.2:

$$D(s1[1..i],s2[1..j]) = \begin{cases} max(i,j), & min(i,j) = 0 \\ min \begin{cases} s1[1..i], s2[1..j-1] + 1 \\ s1[1..i-1], s2[1..j] + 1 \\ s1[1..i-2], s2[1..j-2] + 1 \\ s1[1..i-1],s2[1..j-1] + M(s[i], s[j]) \end{cases} & i,j > 1, s1_{i-1} = s2_j, s1_i = s2_{j-1} \\ min \begin{cases} s1[1..i], s2[1..j-1] + 1 \\ s1[1..i-1], s2[1..j]) + 1 \\ s1[1..i-1],s2[1..j-1] + M(s[i], s[j]) \end{cases} & \text{иначе} \end{cases} \tag{1.2}$$

2 Констукторский раздел

В данном разделе будут рассмотрены схемы алгоритмов, требования к функциональности ПО, и опредены способы тестования.

2.1 Разработка алгоритмов

Ниже представлены схемы алгоритмов поиска растояния Левенштейна:

1) нерекурсивного с заполнением матрицы (2.1);
2) рекурсивного без заполнения матрицы (2.2);
3) рекурсивного с заполнением матрицы (2.3);

и схема алгоритма поиска растояния Дамерау-Левенштейна (2.4).

Рисунок 2.1 — Схема нерекурсивного поиска с заполнением матрицы

Рисунок 2.2 — Схема рекурсивого поиска без заполнения матрицы

Рисунок 2.3 — Схема рекурсивого поиска с заполнением матрицы

Рисунок 2.4 — Схема поиска р. Дамерау-Левенштейна

2.2 Требования к функциональности ПО

В данной работе требуется обеспечить минимальную функциональность консольного приложения:

1) Режим ввода
   а) возможность считать две строки;
   б) вывод расстояний Левенштейна и Дамерау-Левенштейна между строками;
   в) вывод матриц, используемых в вычислении расстояний (если использовались);
2) Режим тестования
   а) вывод таблицы со временем[1] работы и максимальной затраченной памяти каждого из алгоритмов.

По умолчанию приложение работает в режиме ввода, для перехода в режим тестирования необходимо указать ключ -t при запуске.

2.3 Тесты

Тестирование ПО будет проводиться методом чёрного ящика. Необходимо проверить работу системы на тривиальных случаях (одна или обе строки пустые, строки полностью совпадают) и несколько нетривальных случаев.

3 Технологический раздел

В данном разделе будут выбраны средства реплизации ПО и представлен листинг кода.

3.1 Средства реализации

В данной работе используется язык программирования C++, так как язык позволяет написать программу, работающую относительно быстро. Проект выполнен в IDE Visual Studio 2019[2]

Для замера процессорного времени была использован QueryPerformanceCounter[3] из библиотеки WinAPI.

Листинг 3.1 — Функция замера времени

```
using funcDL = std::size_t( * )(const char*, const char* );
double getTime(funcDL getDL, const char* s1, const char* s2, int samples)
{
    LARGE_INTEGER StartingTime, EndingTime, ElapsedMicroseconds;
    LARGE_INTEGER Frequency;

    QueryPerformanceFrequency(&Frequency);
    QueryPerformanceCounter(&StartingTime);

    // Activity to be timed
    for (size_t i = 0; i < samples; i++)
        getDL(s1, s2);

    QueryPerformanceCounter(&EndingTime);
    ElapsedMicroseconds.QuadPart = EndingTime.QuadPart - StartingTime.QuadPart;

    ElapsedMicroseconds.QuadPart *= 1000000;
    ElapsedMicroseconds.QuadPart /= (Frequency.QuadPart * samples);
    return ElapsedMicroseconds.QuadPart;
}
```

3.2 Листинг программы

Ниже представлены листинги кода поиска растояния Левенштейна:

1) нерекурсивного с заполнением матрицы (3.2);
2) рекурсивного без заполнения матрицы (3.3);
3) рекурсивного с заполнением матрицы (3.4);

и код функции поиска растояния Дамерау-Левенштейна (3.5).

Листинг 3.2 — Функция нерекурсивного поиска с заполнением матрицы

```
std::size_t getLevMatr(const char* s1, const char* s2)
```

```
{
        auto n = strlen(s1);
        auto m = strlen(s2);
        auto matr = Matrix(n + 1, m + 1);

        for (size_t i = 0; i <= n; i++)
                matr[i][0] = i;

        for (size_t j = 1; j <= m; j++)
                matr[0][j] = j;

        for (size_t i = 1; i <= n; i++)
        {
                for (size_t j = 1; j <= m; j++)
                {
                        matr[i][j] = _min(_min(
                                matr[i - 1][j] + 1,
                                matr[i][j - 1] + 1),
                                matr[i - 1][j - 1] + (s1[i - 1] != s2[j - 1])
                        );
                }
        }

        return matr[n][m];
}
```

Листинг 3.3 — Функция рекурсивного поиска без заполнения матрицы

```
std::size_t getLevRec(const char* s1, const char* s2)
{
        std::size_t i = strlen(s1);
        std::size_t j = strlen(s2);
        return _getLevRec(s1, i, s2, j);
}
std::size_t _getLevRec(const char* s1, size_t i, const char* s2, size_t j)
{
    std::size_t d;
    if (i == 0)
        d = j;
    else if (j == 0)
        d = i;
    else
    {
        d = _min(_min(
            _getLevRec(s1, i, s2, j - 1) + 1,
            _getLevRec(s1, i - 1, s2, j) + 1),
            _getLevRec(s1, i - 1, s2, j - 1) + (s1[i - 1] != s2[j - 1])
```

```
20          );
21      }
22      return d;
23  }
```

Листинг 3.4 — Функция рекурсивного поиска с заполнением матрицы

```
1   std::size_t getLevRecMatr(const char* s1, const char* s2)
2   {
3       std::size_t n = strlen(s1);
4       std::size_t m = strlen(s2);
5       auto matr = Matrix(n + 1, m + 1);
6       for (size_t i = 0; i < n + 1; i++)
7           for (size_t j = 0; j < m + 1; j++)
8               matr[i][j] = -1;
9       return _getLevRecMatr(s1, n, s2, m, matr);
10  }
11  std::size_t _getLevRecMatr(const char* s1, size_t i, const char* s2, size_t j,
        Matrix& matr)
12  {
13      if (i == 0)
14          matr[i][j] = j;
15      else if (j == 0)
16          matr[i][j] = i;
17      else
18      {
19          if (matr[i][j - 1] == -1)
20              matr[i][j - 1] = _getLevRecMatr(s1, i, s2, j - 1, matr);
21          if (matr[i - 1][j] == -1)
22              matr[i - 1][j] = _getLevRecMatr(s1, i - 1, s2, j, matr);
23          if (matr[i - 1][j - 1] == -1)
24              matr[i - 1][j - 1] = _getLevRecMatr(s1, i - 1, s2, j - 1, matr);
25          matr[i][j] = _min(_min(matr[i][j - 1], matr[i - 1][j]) + 1, matr[i - 1][j -
                1] + (s1[i - 1] != s2[j - 1]));
26      }
27
28      return matr[i][j];
29  }
```

Листинг 3.5 — Функция поиска растояния Дамерау-Левенштейна

```
1   std::size_t getDamLevMatr(const char* s1, const char* s2)
2   {
3       std::size_t n = strlen(s1);
4       std::size_t m = strlen(s2);
5       auto matr = Matrix(n + 1, m + 1);
6
7       for (size_t i = 0; i <= n; i++)
```

```
8           matr[i][0] = i;
9
10      for (size_t j = 1; j <= m; j++)
11          matr[0][j] = j;
12
13      for (size_t i = 1; i <= n; i++)
14      {
15          for (size_t j = 1; j <= m; j++)
16          {
17              matr[i][j] = _min(_min(
18                  matr[i - 1][j] + 1,
19                  matr[i][j - 1] + 1),
20                  matr[i - 1][j - 1] + (s1[i - 1] != s2[j - 1]
21              ));
22
23              if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1])
24                  matr[i][j] = _min(matr[i][j], matr[i - 2][j - 2] + 1);
25          }
26      }
27
28      return matr[n][m];
29  }
```

3.3 Тестирование

№	строка 1	строка 2	ожидаемый результат (р.Л, р.Д-Л)	фактический результат (р.Л, р.Д-Л)
1	0	0	0, 0	0, 0
2	0	ab	2, 2	2, 2
3	abba	baab	3, 2	3, 2
4	abcd	qwer	4, 4	4, 4

Все тесты были пройдены.

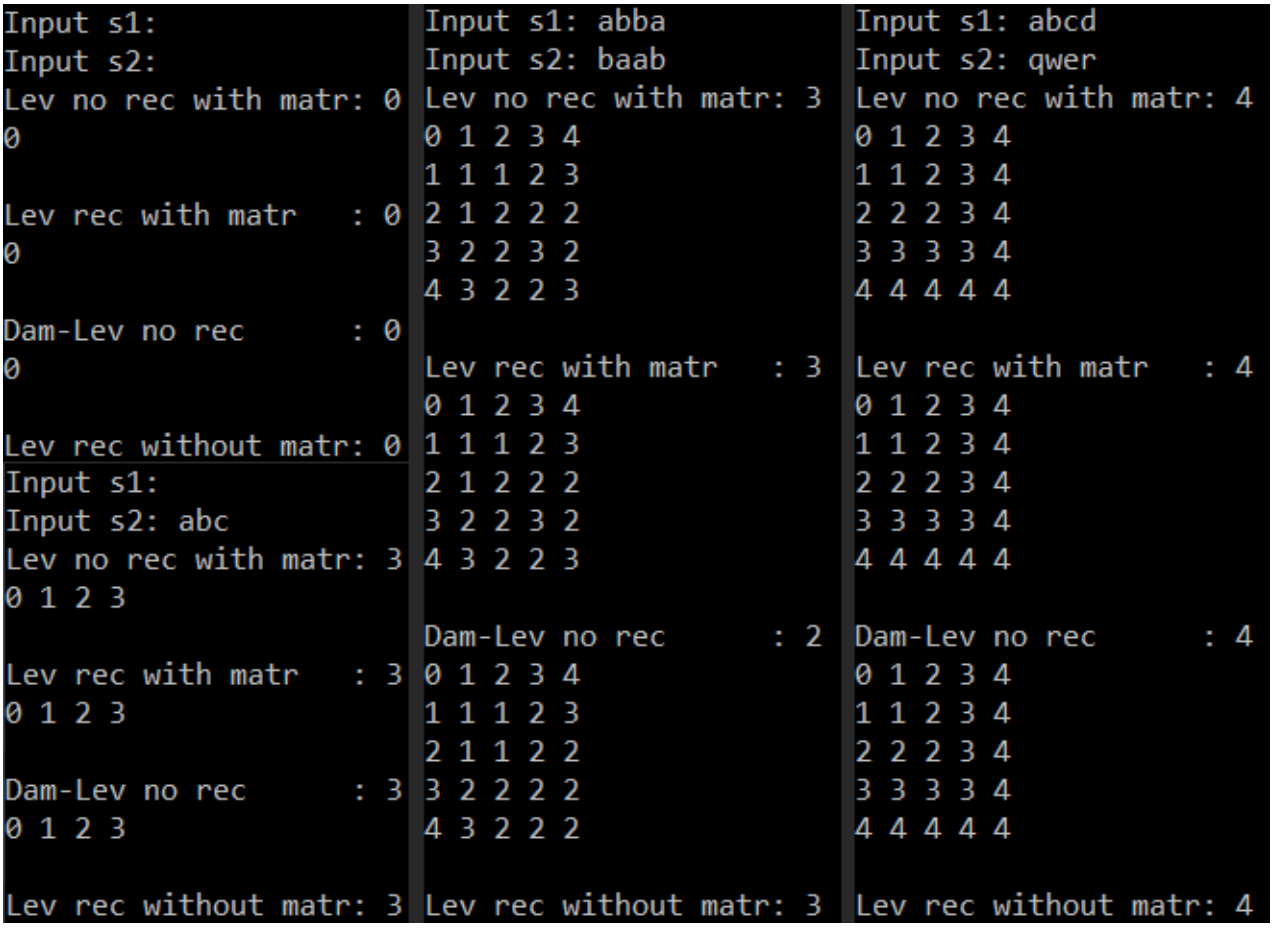

Рисунок 3.1 — Результаты тестирования

4 Экспериментальный раздел

В данном разделе будут проведены эксперименты для проведения сравнительного анализа алгоритмов по затрачиваемому процессорному времени[1] и количеству максимально затрачиваемой памяти.

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведён замер времени работы каждого из алгоритмов:

1) на небольших длинах строк (рисунок 4.1);

2) на относительно больших длинах строк (рисунок 4.2).

Ниже предствалены графики полученных замеров времени (диаграммы 4.3 и 4.4).

Visual studio позволяет оценить затраты динамической памяти, через профилирование heap. Ниже выделение матрицы 10x10

	time (ms)			
len(str)	Dam-Lev no rec	Lev no rec with matr	Lev rec with matr	Lev rec without matr
1	0.417	0.354	0.355	0.043
2	0.514	0.493	0.543	0.133
3	0.484	0.421	0.497	0.597
4	0.845	0.807	0.964	2.980
5	0.921	0.878	1.070	18.331
6	1.011	0.969	1.207	71.911
7	1.121	1.073	1.451	363.163
8	1.244	1.192	1.641	3161.990
9	1.457	1.299	1.896	15813.740
10	1.603	1.213	1.916	52690.635

Рисунок 4.1 — Результаты замера времени на маленьких строках

	time (ms)			
len(str)	Dam-Lev no rec	Lev no rec with matr	Lev rec with matr	Lev rec without matr
1	0.296	0.186	0.314	0.024
51	35.331	10.272	31.266	-1.000
101	108.479	48.553	96.408	-1.000
151	230.661	95.644	277.173	-1.000
201	380.715	177.292	466.735	-1.000
251	627.296	281.747	786.462	-1.000
301	1023.103	380.755	1059.956	-1.000
351	1046.089	800.927	1574.206	-1.000
401	1990.896	973.071	2351.710	-1.000
451	3083.785	1345.672	3157.806	-1.000
501	4465.486	1743.447	3614.684	-1.000
551	5576.579	1961.990	6711.731	-1.000
601	4348.168	2251.130	7197.837	-1.000
651	5652.257	3114.284	6679.272	-1.000
701	5845.979	6119.240	7670.856	-1.000
751	12901.635	3722.417	8825.653	-1.000
801	5967.894	4118.671	11393.080	-1.000
851	6638.438	5513.621	13605.303	-1.000
901	7351.467	6011.998	15329.763	-1.000
951	8078.510	6144.795	15778.976	-1.000
1001	9183.964	6736.900	17986.240	-1.000

Рисунок 4.2 — Результаты замера времени на больших строках

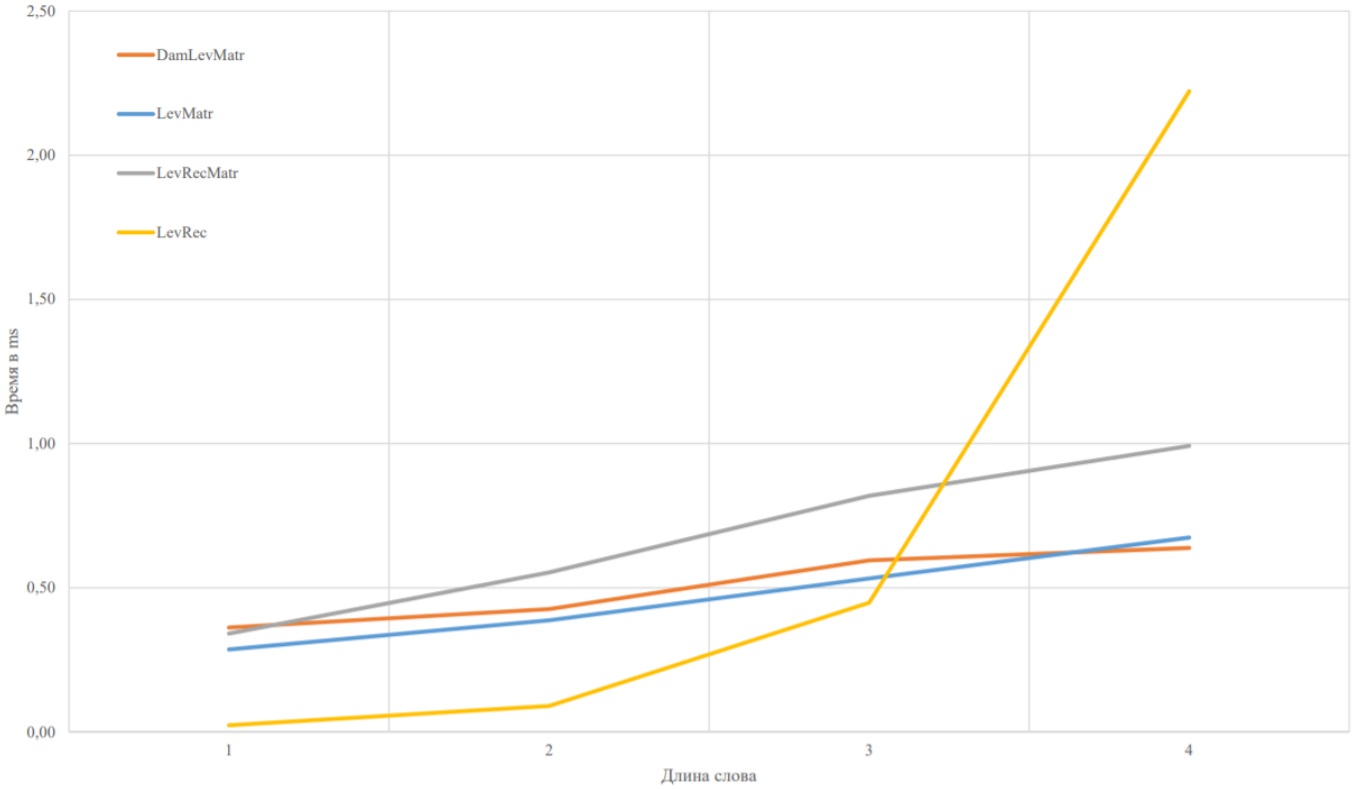

Рисунок 4.3 — Зависимость времени работы алгоритма от длины малых строк

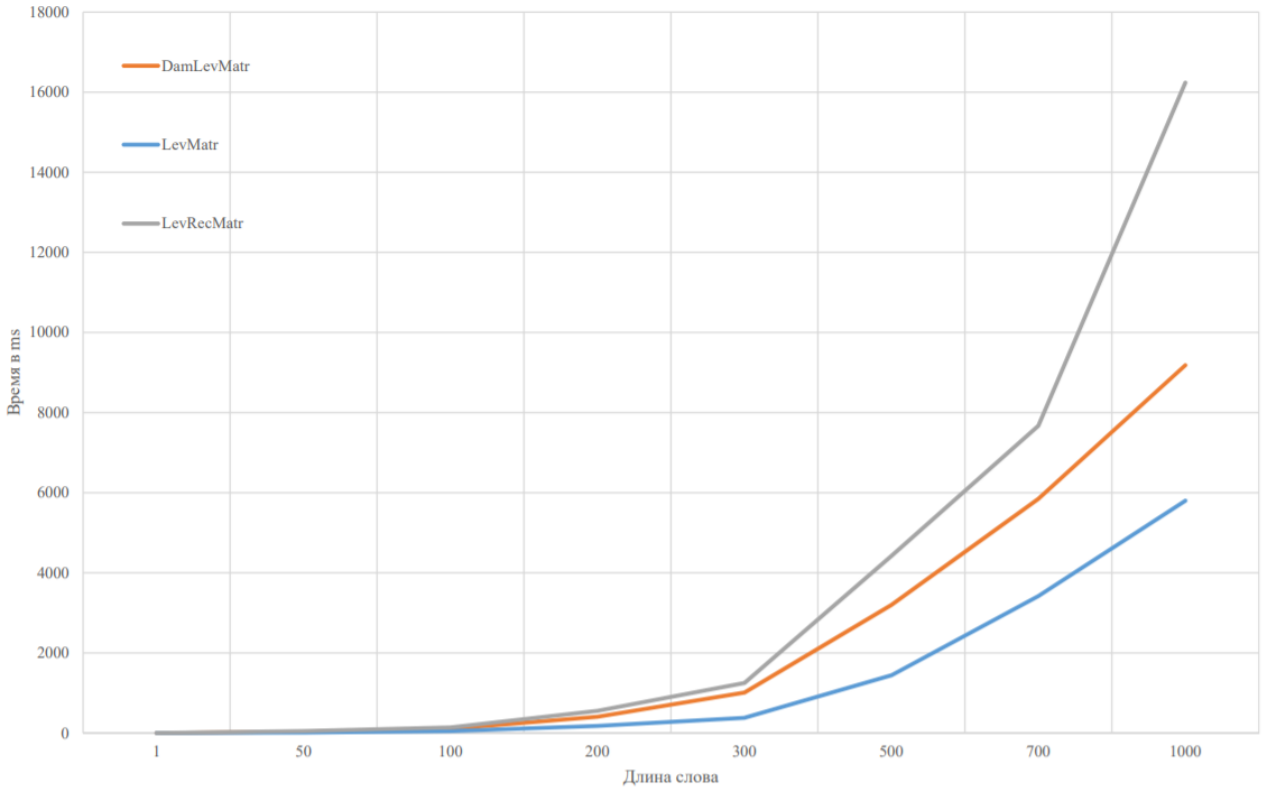

Рисунок 4.4 — Зависимость времени работы алгоритма от длины больших строк

ID	Time	Allocations (Diff)	Heap Size (Diff)
1	13.66s	206 (n/a)	77,77 KB (n/a)
2	13.66s	208 (+2 ↑)	78,27 KB (+0,50 KB ↑)

Заключение

В ходе работы были изучены и реализованы алгоритмы нахождения расстояния Левенштейна (не рекурсивный с заполнением матрицы, рекурсивный без заполнения матрицы, рекурсивный с заполнением матрицы) и Дамерау-Левенштейна (не рекурсивный с заполнением матрицы). Выполнено сравнение перечисленных алгоритмов. В ходе экспериментов было установлено, что алгоритмы использующие матрицы занимают намного больше памяти при обработке длинных строк, чем рекурсивная реализация алгоритма. Изучены зависимости времени выполнения алгоритмов от длины строк. На длинных строках хуже всех себя показала рекурсивная реализация, лучше всех - реализация поиска расстояния Левенштейна не рекурсивно с заполнением матрицы. На маленьких строках (до 3 символов) рекурсивная реализация оказалась быстрее и менее затратна по памяти.

Список использованных источников

1. Time Basics. // [Электронный ресурс]. Режим доступа: https://www.gnu.org/software/libc/manual/html_node/Time-Basics.html, (дата обращения: 11.09.2020).

2. IDE Visual Studio 2019. // [Электронный ресурс]. Режим доступа: https://visualstudio.microsoft.com/ru/vs/, (дата обращения: 11.09.2020).

3. Acquiring high-resolution time stamps. // [Электронный ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/windows/win32/sysinfo/acquiring-high-resolution-time-stamps?redirectedfrom=MSDN, (дата обращения: 11.09.2020).