

*Государственное образовательное учреждение высшего профессионального
образования*

**«Московский государственный технический университет
имени Н. Э. Баумана»
(МГТУ им. Н.Э. Баумана)**

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЁТНО - ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к лабораторной работе на тему:

Рекуррентные соотношения: расстояние между строками

Студент	_____	Киселев А.М.
	(Подпись, дата)	
Преподаватель	_____	Волкова Л.Л.
	(Подпись, дата)	

Москва 2018

Содержание

Введение	3
1 Аналитический раздел	4
1.1 Описание Алгоритмов	4
1.1.1 Расстояние Левенштейна	4
1.1.2 Расстояние Дамерау – Левенштейна	5
2 Конструкторский раздел	7
2.1 Разработка алгоритмов	7
2.1.1 Расстояние Левенштейна(обычный)	7
2.1.2 Расстояние Дамерау – Левенштейна	7
2.1.3 Расстояние Левенштейна(рекурсивный)	7
3 Технологический раздел	9
3.1 Требования к программному ПО	9
3.2 Средства реализации	9
3.3 Листинг	9
4 Экспериментальный раздел	12
4.1 Примеры работы	12
4.2 Постановка эксперимента	13
4.3 Сравнительный анализ на материале экспериментальных данных .	13
Заключение	19

Введение

Целью работы является изучение и применение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна, а так же реализовать алгоритм Левенштейна в рекурсивном виде. Для достижения поставленной цели необходимо решить следующие задачи:

- Изучить алгоритмы Левенштейна и Дамера-Левенштейна нахождения расстояния между строками;
- Применить метод динамического программирования для матричной реализации указанных алгоритмов;
- Получить практические навыки реализации указанных алгоритмов: двух алгоритмов в матричной версии и алгоритма Левенштейна, реализованного рекурсивно;
- Провести сравнительный анализ линейной и рекурсивной реализаций алгоритма Левенштейна по затрачиваемым ресурсам(времени и памяти);
- Привести экспериментальное подтверждение различий во временной эффективности рекурсивной и нерекурсивной реализаций алгоритма Левенштейна при помощи разработанного ПО на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
- Описать и обосновать полученные результаты о выполненной работе;

1 Аналитический раздел

Перед теоритическим изложением алгоритмов, представленных в работе, требуется ввести понятия *редакционного расстояния* и *метода динамического программирования*.

Редакционное расстояние(*расстояние Эйнштейна*) – это минимальное количество редакционных операций, необходимых для преобразования одной строки в другую.

Есть следующие редакционные операции:

- Операции, вес которых - 1:
 - I - insert(вставка);
 - D - delete(удаление);
 - R - replace(замена);
- Операция, вес которой - 0:
 - M - match(совпадение);

Так минимальное расстояние между строками $\min D(\text{'увлечение'}, \text{'развлечение'}) = 3$ (таблица 1.1), но чтобы найти это, требуется перебрать расстояния с разным выравниваем строк по отношению друг к другу.

		у	в	л	е	ч	е	н	и	е
р	а	з	в	л	е	ч	е	н	и	е
I	I	R	M	M	M	M	M	M	M	M

Таблица 1.1 — Пример нахождения редакционного расстояния между строками "увлечение" "развлечение"

Проблема выравнивания решается рекуррентно через расстояния между подстроками фиксированной длины.

1.1 Описание Алгоритмов

Первым появившимся алгоритмом был алгоритм Левенштейна, который заложил фундамент в поиске расстояния между строками.

1.1.1 Расстояние Левенштейна

Расстояние Левенштейна имеет широкую область применения. Алгоритм используется в:

- поисковых системах, в базах данных, в автоматическом распознавании текста и речи для исправления ошибок и опечаток в слове;

- в утилитах для сравнения файлов(таких как diff);
- в биоинформатике для сравнения генов, хромосом и белков;

Алгоритм рекуррентно через расстояния между подстроками i и j находит расстояние между строками $s1$ и $s2$. Проверяя, какое действие будет наиболее выгодным I(insert), D(delete), M(match) или R(replace). Математическое описание данного алгоритма представлено как 1.1 и 1.2.

$$\begin{aligned} D(s1, 0) &= D(s1[0..i], 0) = 0..i, \\ D(0, s2) &= D(0, s2[0..j]) = 0..j \end{aligned} \quad (1.1)$$

$$D(s1, s2) = D(s1[1..i], s2[1..j]) = \min \left(\begin{array}{l} D(s1[1..i], s2[1..j-1]) + 1, \\ D(s1[1..i-1], s2[1..j]) + 1, \\ D(s1[1..i-1], s2[1..j-1]) + \begin{cases} 0, \text{ если } s1[i] = s2[j], \\ 1, \text{ иначе} \end{cases} \end{array} \right) \quad (1.2)$$

,где i – длина подстроки строки $s1$, которая изначально равно длине $s1$, j – длина подстроки строки $s2$, которая изначально равно длине $s2$

Данные вычисления удобно представлять в виде матрицы(пример матрицы 1.2).

	λ	M	Γ
λ	0	1	2
M	1	0	1

Таблица 1.2 — Представление $D('M', 'M\Gamma')$

В виде матрицы редакционные операции можно представить следующим образом:

- I(insert) \rightarrow ;
- D(delete) \downarrow ;
- M(match) or R(replace) \searrow ;

У данного алгоритма есть альтернативная версия, которую придумал Фредерик Дамерау, модифицировав основной алгоритм Левенштейна, который математически можно представить выражениями 1.1, 1.3 и 1.4.

1.1.2 Расстояние Дамерау – Левенштейна

Данный алгоритм применяется также как и обычный в:

- поисковых системах;
- биоинформатике(сравнение белков линейной структуры);

Причиной появления данного алгоритма было огромное количество ошибок ввода – ввод двух соседних символов не в том порядке. Отсюда появляется новая операция в дополнении к уже имеющимся:

- X - exchange(или T - transposition);

Отсюда, формула 1.2 переходит в 1.3:

$$D(s1[1..i], s2[1..j]) = \min \left(\begin{array}{c} D(s1[1..i], s2[1..j-1]) + 1, \\ D(s1[1..i-1], s2[1..j]) + 1, \\ D(s1[1..i-1], s2[1..j-1]) + \begin{array}{c} \left[\begin{array}{c} 0, \text{ если } s1[i] = s2[j], \\ 1, \text{ иначе} \end{array} \right] \end{array} \end{array} \right), \quad (1.3)$$

,где A – выражение 1.4

$$A = \left[\begin{array}{c} D(s1[1..i-2], s2[1..j-2]) + 1, \text{ если } s1[i-1] == s2[j-2] \text{ и } s2[j-1] == s1[i-2] \\ - , \text{ иначе} \end{array} \right] \quad (1.4)$$

Причем последняя операция(X) выполняется, если такая перестановка необходима и требуется, и это не совпадение.

2 Конструкторский раздел

Ниже представлены схемы алгоритмов – Дамерау - Левенштейна и двух реализаций Левенштейна(рекурсивный и обычный).

2.1 Разработка алгоритмов

2.1.1 Расстояние Левенштейна(обычный)

На рисунке 2.1 представлена блок схема алгоритма Левенштейна.

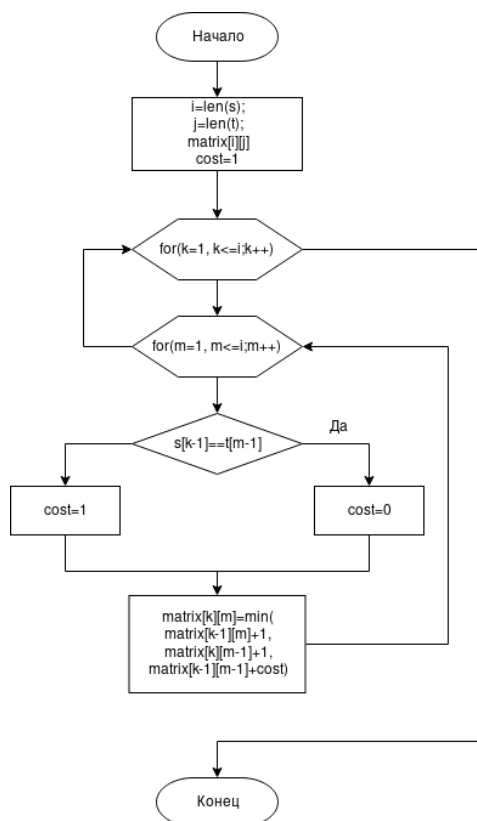


Рисунок 2.1 — Представлена схема алгоритма нахождения расстояния Левенштейна для матричной реализации

2.1.2 Расстояние Дамерау – Левенштейна

На рисунке 2.2 представлена блок схема алгоритма Дамерау – Левенштейна.

2.1.3 Расстояние Левенштейна(рекурсивный)

На рисунке 2.3 представлена блок схема рекурсивного алгоритма Левенштейна, в реализации которого используется функция $\text{int match}(\text{char } c, \text{char } d)$, которая возвращает 0, если символы c и d совпадают, иначе 1.

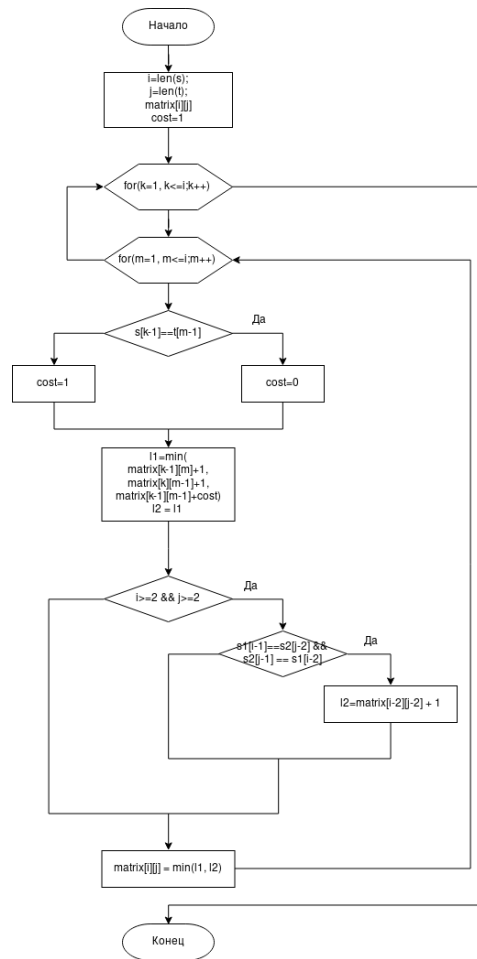


Рисунок 2.2 — Представлена схема алгоритма нахождения расстояния Дamerau – Левенштейна для матричной реализации

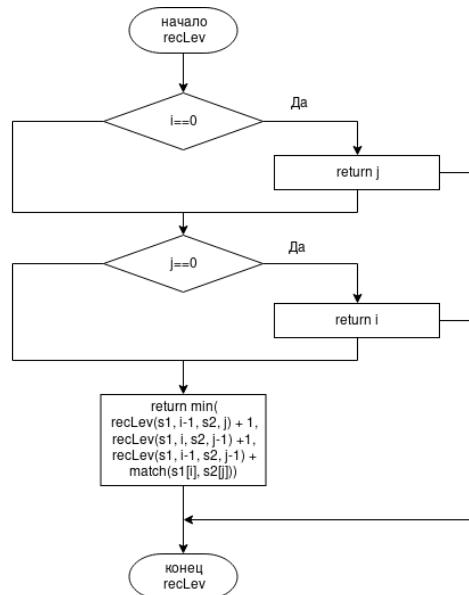


Рисунок 2.3 — Представлена схема алгоритма нахождения расстояния Левенштейна для рекурсивной реализации

3 Технологический раздел

Данный раздел содержит информацию о реализации ПО и листингах программ.

3.1 Требования к программному ПО

Программы реализованы в двух вариациях. Первая – каждая программа реализована поотдельности и обладает своим собственным функционалом – возможность вводить строки и получать вывод в поток вывода. Вторая – программа, содержащая в себе функции с реализованными алгоритмами, в которой производится тестирование времени алгоритмов и вывод отправляется в поток вывода информации.

3.2 Средства реализации

Данные реализации разрабатывались на языке C, использовался компилятор gcc-8.2.1. Замер времени производится с помощью библиотеки *time.h*. Данная библиотека позволяет производить замер тиков, откуда можно получать время в секундах.

3.3 Листинг

Алгоритм Левенштейна представлен в листинге 3.1

```
1 int levensteinDistance( char* s1, char* s2,
2                          int ( * matrix) [MAX_STRING_SIZE] [MAX_STRING_SIZE] )
3 {
4     size_t len1 = strlen(s1);
5     size_t len2 = strlen(s2);
6
7     matrixInit(matrix, len1, len2);
8
9     int cost = 1;
10    for(int i = 1; i <= len1; i++)
11        for(int j = 1; j <= len2; j++) {
12            if(s1[i-1] == s2[j-1])
13                cost = 0;
14            else
15                cost = 1;
16            ( * matrix) [i] [j] = min( 3,
17                                     ( * matrix) [i-1] [j]+1,
18                                     ( * matrix) [i] [j-1]+1,
19                                     ( * matrix) [i-1] [j-1]+cost );
20        }
21 }
```

Листинг 3.1 — Levenstein algorithm

Листинг алгоритма Дамерау – Левенштейна представлен в листинге 3.2

```
1 int levensteinDistance( char* s1, char* s2,
2                         int ( * matrix)[MAX_STRING_SIZE][MAX_STRING_SIZE] )
3 {
4     size_t len1 = strlen(s1);
5     size_t len2 = strlen(s2);
6
7     matrixInit(matrix, len1, len2);
8
9     int cost = 1;
10    for(int i = 1; i <= len1; i++)
11        for(int j = 1; j <= len2; j++) {
12            int flag = 0;
13            if(s1[i-1] == s2[j-1])
14                cost = 0;
15            else
16                cost = 1;
17            int l1= min( 3,
18                        ( * matrix)[i-1][j]+1,
19                        ( * matrix)[i][j-1]+1,
20                        ( * matrix)[i-1][j-1]+cost );
21            int l2 = l1;
22            if(i>=2 && j>=2)
23                if(s1[i-1] == s2[j-2] && s2[j-1] == s1[i-2])
24                    l2 = ( * matrix)[i-2][j-2]+1;
25            ( * matrix)[i][j] = min(2, l1, l2);
26        }
27 }
```

Листинг 3.2 — Damerau – Levenstein algorithm

Листинг алгоритма Левенштейна в рекурсивной реализации представлен в листинге 3.3

```
1 int levensteinDistance( char* s1, int i, char* s2, int j )
2 {
3     if(i == 0) return j;
4     if(j == 0) return i;
5
6     return min( 3,levensteinDistance(s1, i-1, s2, j) + 1,
7                 levensteinDistance(s1, i, s2, j-1) + 1,
8                 levensteinDistance(s1, i-1, s2, j-1) + match(s1[i], s2[j]
9     ));
10 }
```

Листинг 3.3 — Recursive Levenstein algorithm

Листинг функции match() представлен в листинге 3.4

```
1 int match(char c, char d)
2 {
3     if(c == d) return 0;
4     else return 1;
5 }
```

Листинг 3.4 — Function match()

4 Экспериментальный раздел

Данный раздел посвящен тестированию и исследованию трех реализованных алгоритмов: алгоритмам Левенштейна, Дамерау – Левенштейна и рекурсивной реализации Левенштейна.

4.1 Примеры работы

Алгоритм Левенштейна:

```
1 Input s1: ОТАРА
2 Input s2: ТАРТАР
3
4 0 1 2 3 4 5 6
5 1 1 2 3 4 5 6
6 2 1 2 3 3 4 5
7 3 2 1 2 3 3 4
8 4 3 2 1 2 3 3
9 5 4 3 2 2 2 3
```

```
1 Input s1: MGTU
2 Input s2: MTGU
3
4 0 1 2 3 4
5 1 0 1 2 3
6 2 1 1 1 2
7 3 2 1 2 2
8 4 3 2 2 2
```

Алгоритм Дамерау – Левенштейна:

```
1 Input s1: MGTU
2 Input s2: MTGU
3
4 0 1 2 3 4
5 1 0 1 2 3
6 2 1 1 1 2
7 3 2 1 1 2
8 4 3 2 2 1
```

```
1 Input s1: KNUTH
2 Input s2: TANENBAUM
3
4 0 1 2 3 4 5 6 7 8 9
```

5	1	1	2	3	4	5	6	7	8	9
6	2	2	2	2	3	4	5	6	7	8
7	3	3	3	3	3	4	5	6	6	7
8	4	3	4	4	4	4	5	6	7	7
9	5	4	4	5	5	5	5	6	7	8

Алгоритм Левенштейна(Рекурсивный):

1	Input s1: MGTU
2	Input s2: MGTU
3	Length is : 2

1	Input s1: OTAPA
2	Input s2: TAPТАP
3	Length is : 3

4.2 Постановка эксперимента

Эксперимент проводится в виде поочередного запуска программы, в которой находится расстояние между строками, подающимися на вход. Длина строк варьируется от 100 до 1000 с шагом 100, при чем каждую итерацию производится 100 кратный пересчет расстояния с одинаковыми данными для частоты эксперимента. Далее находится среднее значение времени для каждой подсчитанной части. Так как частота вызовов рекурсивной функции становится большой начиная со строк длиной более 6 символов, очень сложно реализовать эксперимент с длиной строк от 100 до 1000 на этом алгоритме, так как вычислительная машина, на которой проводится эксперимент не достаточно мощная. Поэтому для рекурсивной реализации будет проводиться эксперимент с длиной строк от 1 до 10 с шагом 1.

4.3 Сравнительный анализ на материале экспериментальных данных

На рисунке 4.1 представлен первый эксперимент, на котором наглядно продемонстрировано изменение времени двух алгоритмов: Дамерау – Левенштейна и Левенштейна. Рекурсивная реализация отсутствует(причины представлены во втором эксперименте 4.2). Здесь можно заметить, что реализация Дамерау – Левенштейна начинает значительно медленнее работать, чем реализация Левенштейна.

Рисунок 4.2 отображает второй эксперимент, в котором уже в расчет брался рекурсивный метод Левенштейна. Он не был включен в первый, потому что время, затраченное на нахождение расстояния резко подскакивает уже на длинных строк равных 5. Чем больше значение, тем больше время(оно начинает повышаться в несколь-

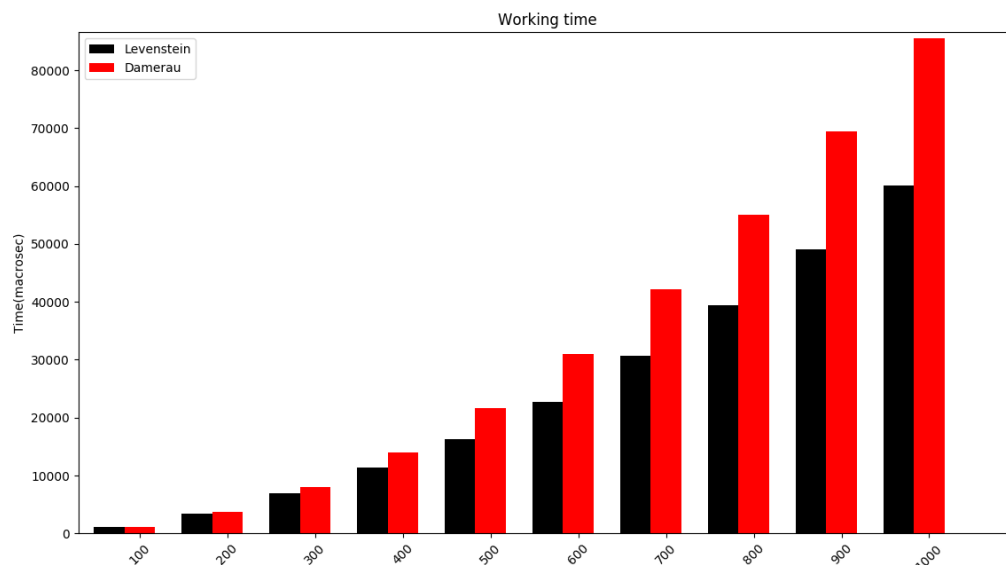


Рисунок 4.1 — Эксперимент первый, демонстрация различия времени для алгоритмов Левенштейна и Дамерау – Левенштейна

ко раз). Если обратить внимание на Левенштейна и Дамерау – Левенштейна, можно заметить, что алгоритм Левенштейна работает медленнее Дамерау – Левенштейна с маленькими строками, но в первом эксперименте наглядно показано, что на больших строках, Левенштейн показывает большую эффективность.

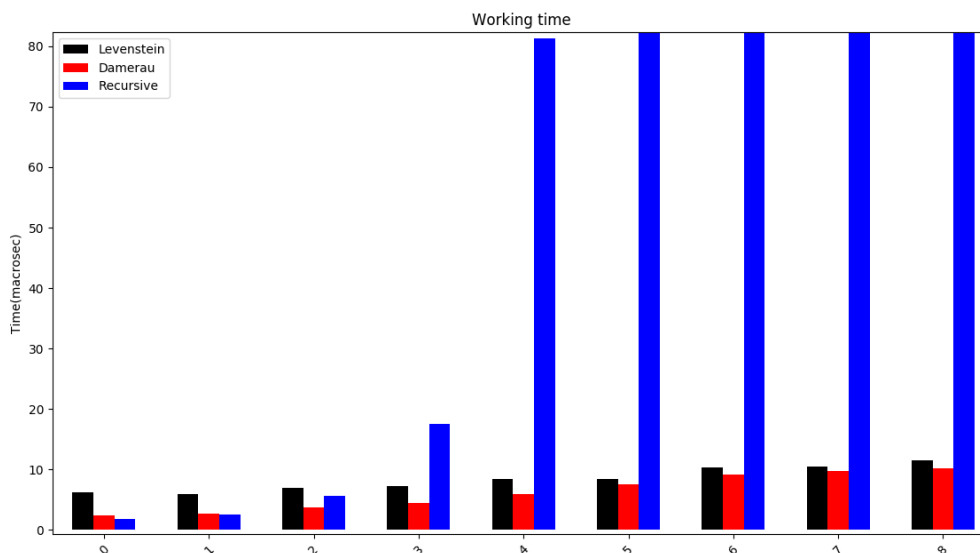


Рисунок 4.2 — Эксперимент второй, демонстрация различия времени для алгоритмов Левенштейна и Дамерау – Левенштейна и рекурсивной реализации Левенштейна

Из-за большого количества рекурсивных запросов, метод работает медленно и чем больше длина слова, тем больше вызовов производится:

```
1   Input s1: OTAPA
2   Input s2: TAPTAP
3   Length is : 3
4   function called 5479 times
```

```
1   Input s1: FLEICHE
2   Input s2: EUROPA
3   Length is : 6
4   function called 29737 times
```

```
1   Input s1: ENTSCULDIGUNG
2   Input s2: EINWAGEN
3   Length is : 11
4   function called 21820057 times
```

В листинге 4.1 представлено рекурсивное дерево на примере поиска редакционного расстояния между строками MGT, MTG. Производится очень много повторяющихся вызовов, которые каждый раз пересчитываются.

```
1   function called 94 times
2   (MGT, MTG)
3   '--(MG, MT)
4   |   '--(M, M)
5   |   |   '--(, )
6   |   |   '--(M, )
7   |   |   '--(, M)
8   |   '--(MG, M)
9   |   |   '--(M, )
10  |   |   '--(MG, )
11  |   |   '--(M, M)
12  |   |       '--(, )
13  |   |       '--(M, )
14  |   |       '--(, M)
15  |   '--(M, MT)
16  |       '--(, M)
17  |       '--(M, M)
18  |       |   '--(, )
19  |       |   '--(M, )
20  |       |   '--(, M)
21  |       '--(, MT)
```

22		‘--(MGT, MT)
23		‘--(MG, M)
24		‘--(M,)
25		‘--(MG,)
26		‘--(M, M)
27		‘--(,)
28		‘--(M,)
29		‘--(, M)
30		‘--(MGT, M)
31		‘--(MG,)
32		‘--(MGT,)
33		‘--(MG, M)
34		‘--(M,)
35		‘--(MG,)
36		‘--(M, M)
37		‘--(,)
38		‘--(M,)
39		‘--(, M)
40		‘--(MG, MT)
41		‘--(M, M)
42		‘--(,)
43		‘--(M,)
44		‘--(, M)
45		‘--(MG, M)
46		‘--(M,)
47		‘--(MG,)
48		‘--(M, M)
49		‘--(,)
50		‘--(M,)
51		‘--(, M)
52		‘--(M, MT)
53		‘--(, M)
54		‘--(M, M)
55		‘--(,)
56		‘--(M,)
57		‘--(, M)
58		‘--(, MT)
59		‘--(MG, MTG)
60		‘--(M, MT)
61		‘--(, M)
62		‘--(M, M)
63		‘--(,)
64		‘--(M,)
65		‘--(, M)
66		‘--(, MT)
67		‘--(MG, MT)
68		‘--(M, M)


```

69 | | '---(, )
70 | | '---(M, )
71 | | '---(, M)
72 | | '---(MG, M)
73 | | '---(M, )
74 | | '---(MG, )
75 | | '---(M, M)
76 | | '---(, )
77 | | '---(M, )
78 | | '---(, M)
79 | | '---(M, MT)
80 | | '---(, M)
81 | | '---(M, M)
82 | | | '---(, )
83 | | | '---(M, )
84 | | | '---(, M)
85 | | | '---(, MT)
86 | | '---(M, MTG)
87 | | | '---(, MT)
88 | | | '---(M, MT)
89 | | | '---(, M)
90 | | | '---(M, M)
91 | | | | '---(, )
92 | | | | '---(M, )
93 | | | | '---(, M)
94 | | | | '---(, MT)
95 | | | | '---(, MTG)

```

Листинг 4.1 — Recursive tree example MGT, MTG

Из диаграмм наглядно видно, что алгоритм Левенштейна гораздо эффективнее его рекурсивной реализации, при работе с большим объемом данных рекурсивная реализация просто никуда не годится.

Память в программах была использована эффективно, не было никаких утечек:

Рекурсивная реализация алгоритма Левенштейна(valgrind):

```

1 ==21413== HEAP SUMMARY:
2 ==21413==      in use at exit: 0 bytes in 0 blocks
3 ==21413==    total heap usage: 2 allocs , 2 frees , 2,048 bytes allocated

```

Алгоритм Левенштейна(valgrind):

```

1 ==21460== HEAP SUMMARY:
2 ==21460==      in use at exit: 0 bytes in 0 blocks
3 ==21460==    total heap usage: 2 allocs , 2 frees , 2,048 bytes allocated

```

Алгоритм Дамерау – Левенштейна(valgrind):

```
1 ==21489== HEAP SUMMARY:
2 ==21489==      in use at exit: 0 bytes in 0 blocks
3 ==21489==    total heap usage: 2 allocs , 2 frees , 2,048 bytes allocated
```

Заключение

В результате выполнения задания были получены следующие основные результаты:

- изучены теоретические понятия редукционного расстояния
- изучены основные алгоритмы нахождения редукционного расстояния между строками: алгоритм Левенштейна, Дамерау – Левенштейна
- проведен аналитический вывод формул для заполнения матриц расстояний
- проведено сравнение трех реализаций заданных алгоритмов, были выявлены их слабые места
- в рамках данной работы было выяснено, что рекурсивный алгоритм работает намного медленнее двух других из-за большого количества рекурсивных вызовов, что делает его бесполезным при работе с большими объемами информации, а алгоритм Левенштейна работает медленнее Дамерау – Левенштейна на маленьких словах, но на словах больше 6 символов, Дамерау – Левенштейн начинает работать медленнее