



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: Динамическое программирование

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б
(Группа)

Н. В. Ляпина
(Подпись, дата) (И.О. Фамилия)

Преподаватель

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

Москва, 2021

Содержание

Введение	3
1 Аналитический раздел	4
1.1 Расстояние Левенштейна	4
1.2 Нерекursивный алгоритм нахождения расстояния Левенштейна с использованием матрицы	4
1.3 Расстояние Дамерау-Левенштейна	5
1.4 Нерекursивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием матрицы	5
1.5 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна	6
1.6 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием матрицы (кэшированием)	6
2 Конструкторский раздел	7
2.1 Разработка алгоритмов	7
2.2 Требования к функциональности ПО	12
2.3 Тесты	12
3 Технологический раздел	13
3.1 Средства реализации	13
3.2 Листинг программы	13
3.3 Тестирование	17
3.4 Сравнительный анализ потребляемой памяти	18
4 Экспериментальный раздел	20
4.1 Сравнительный анализ на основе замеров времени работы алгоритмов	20
4.2 Вывод	21
Заключение	22
Список источников	23

Введение

Термин «динамическое программирование» впервые был использован в середине 20-го века. С тех пор им ежедневно пользуется каждый пользователь Интернета, даже не подозревая об этом.

Динамическое программирование – это способ решения сложных задач путём разбиения их на более простые подзадачи. Так, например, одной из задач динамического программирования является задача сравнения двух строк. Советский математик Владимир Иосифович Левенштейн ввел понятие «рекадционного расстояния» между двумя строковыми последовательностями и разработал алгоритм для его нахождения.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- 1) автоматического исправления ошибок в слове;
- 2) сравнение введённых слов со словарями в поисковых запросах;
- 3) в биоинформатике для сравнения генов, хромосом и белков.

Цель лабораторной работы:

изучение и исследование особенностей задач динамического программирования.

Задачи лабораторной работы:

- 1) выбрать инструменты для замера процессорного времени выполнения реализации алгоритмов;
- 2) изучить алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна;
- 3) реализовать:
 - а) нерекурсивный метод поиска расстояния Левенштейна;
 - б) нерекурсивный метод поиска расстояния Дамерау-Левенштейна;
 - в) рекурсивный метод поиска расстояния Дамерау-Левенштейна;
 - г) рекурсивный с кэшированием метод поиска расстояния Дамерау-Левенштейна;
- 4) замерить процессорное время и потребленную память для всех реализованных алгоритмов;
- 5) провести анализ работы программы по времени и по памяти, выяснить влияющие на них характеристики.

1 Аналитический раздел

1.1 Расстояние Левенштейна

Расстояние Левенштейна - это минимальное необходимое количество редакторских операций (вставки, замены, удаления) для преобразования одной строки в другую. Все три операции обладают штрафом 1. Операция совпадения обладает штрафом 0.

Обозначение редакторских операций:

- I (insert) – вставка символа;
- R (replace) – замена символа;
- D (delete) – удаление символа;
- M (match) – совпадение символов.

Задача по нахождению Расстояния Левенштейна заключается в поиске минимального количества операций для превращения одной строки в другую.

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

$$D_{s_1, s_2}(i, j) = \begin{cases} \max(i, j), & \text{если } \min(i, j) = 0, \\ \min\{ \\ D_{s_1, s_2}(i, j - 1) + 1, \\ D_{s_1, s_2}(i - 1, j) + 1, \\ D_{s_1, s_2}(i - 1, j - 1) + m(a[i], b[j]) \} \end{cases}, \quad (1.1)$$

где i, j – индексы i-го и j-го символов строк s_1 и s_2 соответственно.

1.2 Нерекурсивный алгоритм нахождения расстояния Левенштейна с использованием матрицы

Данный алгоритм заключается в решении задачи с использованием матрицы расстояний, которая построчно заполняется последовательно вычисляемыми $D(i, j)$.

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

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

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

1.3 Расстояние 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 N и M соответственно) в некотором алфавите V , тогда расстояние Дамерау-Левенштейна можно подсчитать по рекуррентной формуле (1.2):

$$D_{a,b}(i, j) = \begin{cases} \max(i, j), & \text{если } \min(i, j) = 0, \\ \min\{ \\ \quad D_{a,b}(i, j - 1) + 1, \\ \quad D_{a,b}(i - 1, j) + 1, \\ \quad D_{a,b}(i - 1, j - 1) + m(a[i], b[j]), & \text{иначе} \\ \quad \left[\begin{array}{ll} D_{a,b}(i - 2, j - 2) + 1, & \text{если } i, j > 1; \\ & a[i] = b[j - 1]; \\ & b[j] = a[i - 1] \\ & \infty, & \text{иначе} \end{array} \right. \\ \} \end{cases}, \quad (1.2)$$

где i, j — индексы i -го и j -го символов строк s_1 и s_2 соответственно.

1.4 Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна с использованием матрицы

Данный алгоритм, также как и алгоритм Левенштейна, заключается в решении задачи с использованием матрицы расстояний, которая построчно заполняется последовательно вычисляемыми $D(i, j)$.

В Таблице 1.2 минимальное расстояние между словом "тест" и "тетс" равно 1. Последовательность редакторских операций, которая привела к ответу - ММХ.

Таблица 1.2 — Пример работы преобразования слова "тетс" в "тест"

	λ	Т	Е	Т	С
λ	0	1	2	3	4
Т	1	0	1	2	3
Е	2	1	0	1	2
С	3	2	1	1	2
Т	4	3	2	2	1

Голубым и красным цветом выделены клетки, задействованные в операции обмена Х.

1.5 Рекурсивный алгоритм нахождения расстояния Дameraу-Левенштейна

Суть рекурсивного алгоритма заключается в реализации формулы 1.2.

1.6 Рекурсивный алгоритм нахождения расстояния Дameraу-Левенштейна с использованием матрицы (кэшированием)

Для оптимизации рекурсивного алгоритма нахождения расстояния Дameraу-Левенштейна допустимо добавить матрицу для хранения значений $D(i, j)$ для того, чтобы не вычислять их заново раз за разом. Таким образом, при обработке ещё не затронутых данных, результат нахождения расстояния будет занесён в так называемую матрицу расстояний. В ином случае, если для рассматриваемого случая информация о расстоянии уже имеется в матрице, алгоритм будет переходить к следующему шагу.

Вывод

Для каждого рассмотренного алгоритма имеется некоторая рекуррентная формула, что даёт возможность изучить как рекурсивные, так и итеративные реализации алгоритмов. Для оптимизации рекурсивных алгоритмов в рассмотрение вводится матрица, в которую записываются все промежуточные вычисленные значения. Эта же матрица применяется и при реализации итеративных алгоритмов.

2 Конструкторский раздел

Алгоритмы нахождения расстояния Левенштейна и Дameraу-Левенштейна можно реализовать несколькими способами. В данном разделе будут рассмотрены схемы алгоритмов, требования к функциональности ПО, и определены способы тестирования.

2.1 Разработка алгоритмов

Ниже будут представлены схемы алгоритмов поиска расстояния Дameraу-Левенштейна:

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

Также будет представлена схема нерекурсивного алгоритма поиска расстояния Левенштейна (рисунок 2.4).

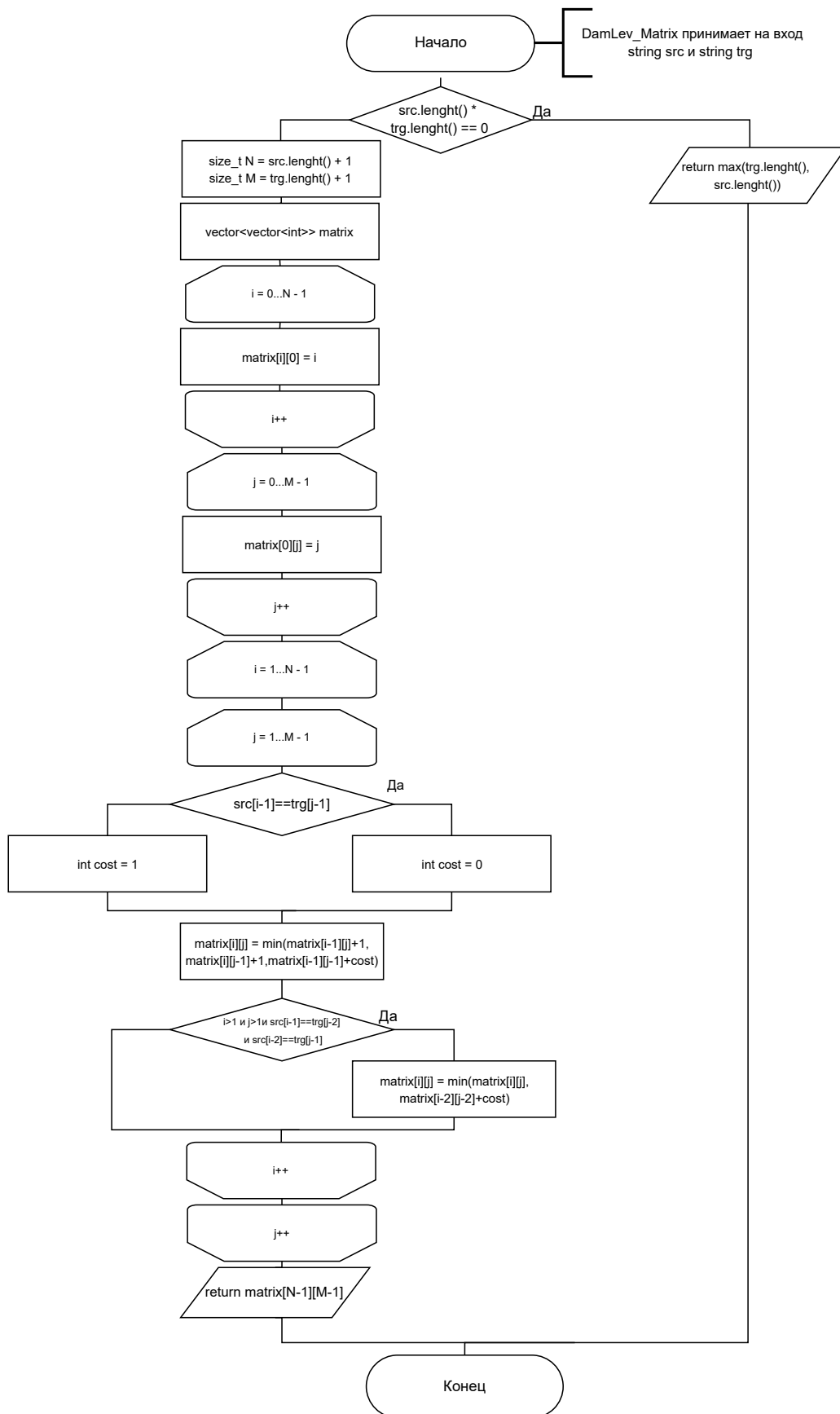


Рисунок 2.1 — Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

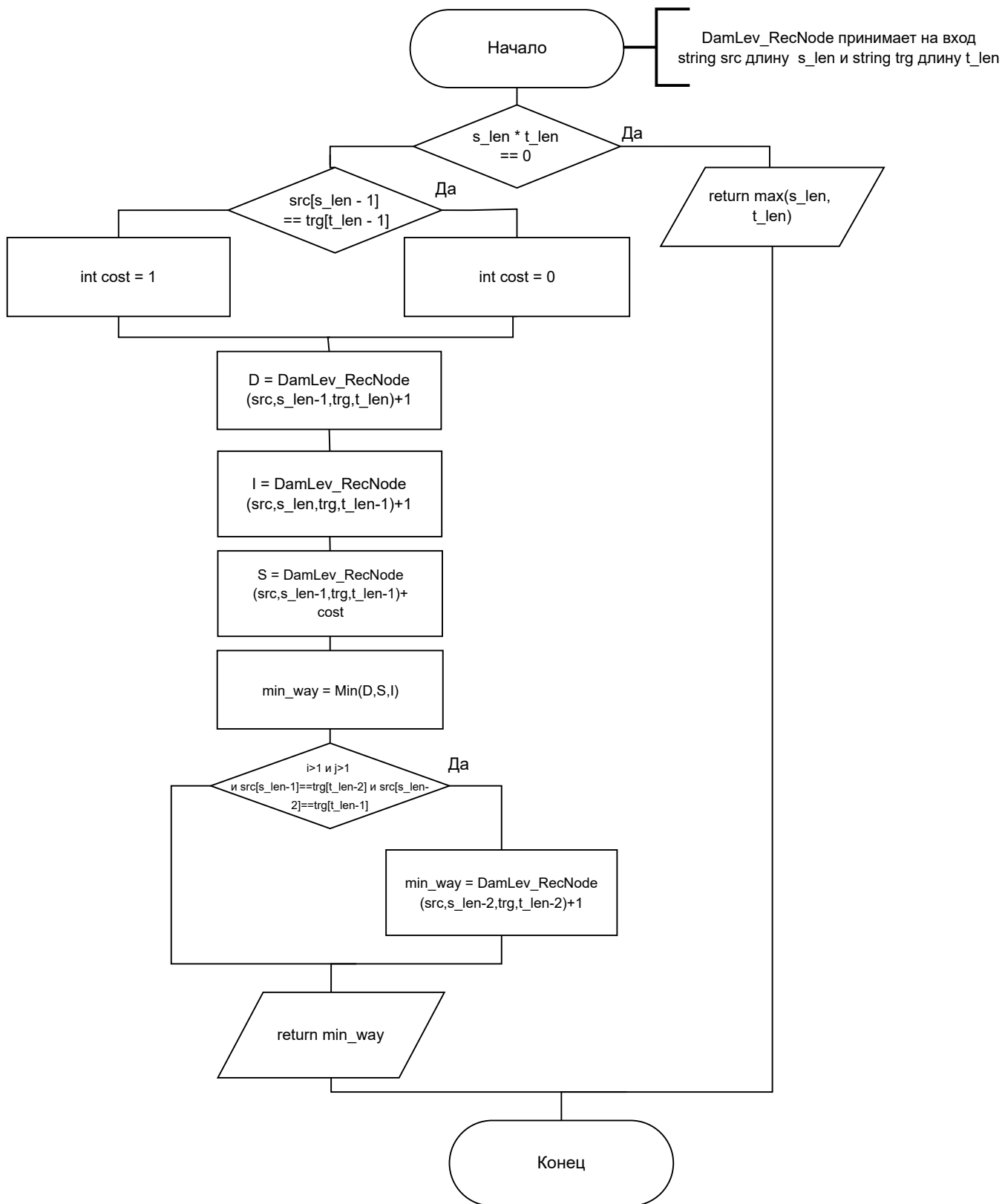


Рисунок 2.2 — Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

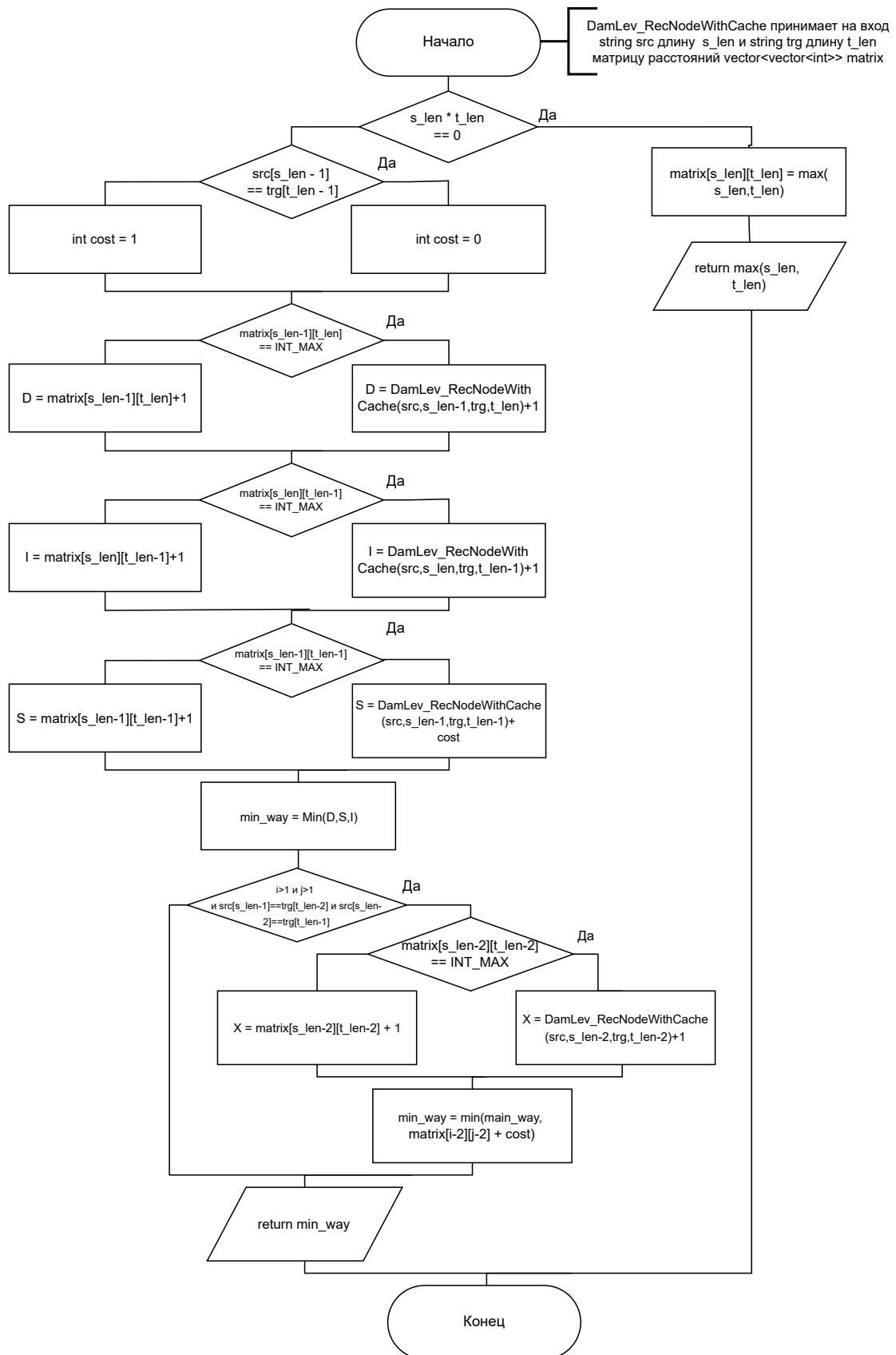


Рисунок 2.3 — Схема рекурсивного алгоритма нахождения расстояния Дameraу-Левенштейна с кэшированием

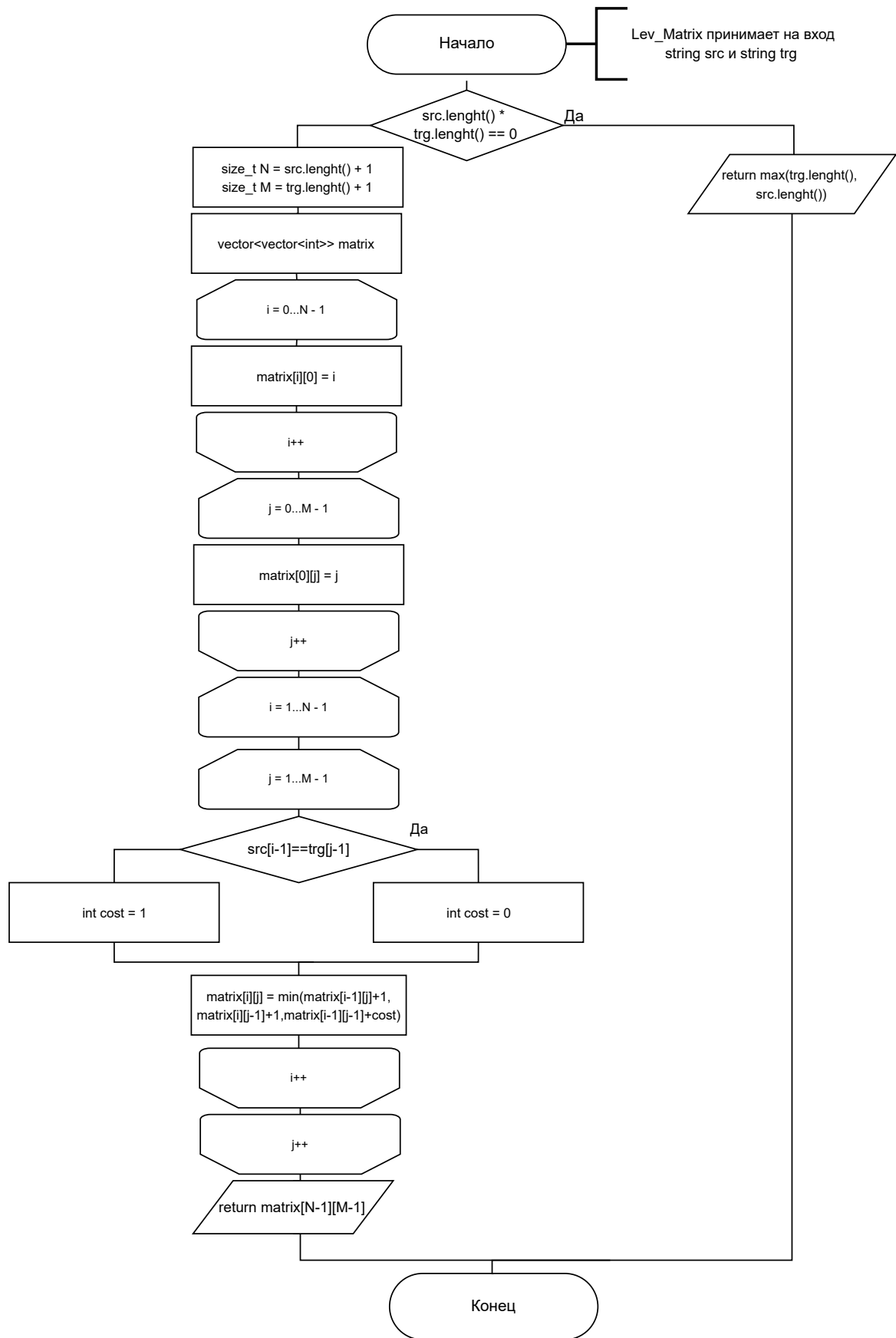


Рисунок 2.4 — Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

2.2 Требования к функциональности ПО

В данной работе требуется обеспечить следующую функциональность.

- 1) Пользовательский режим:
 - а) возможность считать две строки;
 - б) вывод расстояний Левенштейна и Дамерау-Левенштейна между строками.
- 2) Тестовый режим:
 - а) возможность замера процессорного времени реализации каждого алгоритма для строк, заданных внутри программы;
- 3) Экспериментальный режим:
 - а) вывод графиков с процессорным временем работы алгоритмов для варьирующихся строк.

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 нетривиальных случаев.

Вывод

В данном разделе были разработаны схемы используемых алгоритмов и требования к функциональности программы.

3 Технологический раздел

В данном разделе представлены средства, использованные в процессе разработки для реализации задачи, а также листинг кода программы. Кроме того показаны результаты тестирования и анализа затрачиваемой памяти.

3.1 Средства реализации

Для реализации поставленной задачи был использован язык C++, так как имеется большой опыт работы с ним. Для измерения процессорного времени была использована ассемблерная вставка.

3.2 Листинг программы

В приведенных ниже листингах представлены следующие реализации:

- 1) нерекурсивный алгоритм нахождения расстояния Левенштейна с использованием матрицы (листинг 3.1 3.2);
- 2) нерекурсивный алгоритм нахождения расстояния Дameraу-Левенштейна с использованием матрицы (листинг 3.3 3.4);
- 3) рекурсивный алгоритм нахождения расстояния Дameraу-Левенштейна (листинг 3.5);
- 4) рекурсивный алгоритм нахождения расстояния Дameraу-Левенштейна с использованием матрицы (кэшированием) (листинг 3.6 3.7).

Листинг 3.1 — Функция поиска расстояния Левенштейна нерекурсивно

```
1
2 int Lev_Matrix(string src , string trg)
3 {
4     if (src.length() * trg.length() == 0)
5     {
6         return max(trg.length() , src.length());
7     }
8
9     size_t N = src.length() + 1;
10    size_t M = trg.length() + 1;
11
12    vector<vector<int>> matrix (N);
13
14    for (size_t i = 0; i < N; i++)
15        matrix[i].resize(M);
16
17    for (size_t i = 0; i < N; i++)
18        matrix[i][0] = i;
```

Листинг 3.2 — Функция поиска расстояния Левенштейна нерекурсивно

```

1   for (size_t j = 0; j < M; j++)
2       matrix[0][j] = j;
3
4   for (size_t i = 1; i < N; i++)
5   {
6       for (size_t j = 1; j < M; j++)
7       {
8           int cost = src[i - 1] == trg[j - 1] ? 0 : 1;
9           matrix[i][j] = CheckMin(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1,
10                                   matrix[i - 1][j - 1] + cost);
11       }
12   }
13   return matrix[N - 1][M - 1];
14 }
```

Листинг 3.3 — Функция поиска расстояния Дамерау-Левенштейна с использованием матрицы

```

1
2   int DamLev_Matrix(string src, string trg)
3   {
4       if (src.length() * trg.length() == 0)
5       {
6           return max(trg.length(), src.length());
7       }
8
9       size_t N = src.length() + 1;
10      size_t M = trg.length() + 1;
11
12      vector<vector<int>> matrix (N);
13
14      for (size_t i = 0; i < N; i++)
15          matrix[i].resize(M);
16
17      for (size_t i = 0; i < N; i++)
18          matrix[i][0] = i;
19
20      for (size_t j = 0; j < M; j++)
21          matrix[0][j] = j;
22
23      for (size_t i = 1; i < N; i++)
24      {
```

Листинг 3.4 — Функция поиска расстояния Дамерау-Левенштейна с использованием матрицы

```

1      for (size_t j = 1; j < M; j++)
2      {
3          int cost = src[i - 1] == trg[j - 1] ? 0 : 1;
4          matrix[i][j] = CheckMin(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1,
                                   matrix[i - 1][j - 1] + cost);
5
6          if (i > 1 && j > 1 && src[i - 1] == trg[j - 2] && src[i - 2] == trg[j -
            1])
7              matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + cost);
8      }
9  }
10 return matrix[N - 1][M - 1];
11 }

```

Листинг 3.5 — Функция рекурсивного поиска расстояния Дамерау-Левенштейна

```

1
2 static int DamLev_RecNode(string src, size_t s_len, string trg, size_t t_len)
3 {
4     if (s_len * t_len == 0)
5         return max(s_len, t_len);
6
7     int cost = src[s_len - 1] == trg[t_len - 1] ? 0 : 1;
8
9     int D = DamLev_RecNode(src, s_len - 1, trg, t_len) + 1;
10    int I = DamLev_RecNode(src, s_len, trg, t_len - 1) + 1;
11    int S = DamLev_RecNode(src, s_len - 1, trg, t_len - 1) + cost;
12
13    int min_way = CheckMin(D, I, S);
14
15    if (s_len > 1 && t_len > 1 && src[s_len - 1] == trg[t_len - 2] && src[s_len - 2]
        == trg[t_len - 1])
16        min_way = min(min_way, DamLev_RecNode(src, s_len - 2, trg, t_len - 2) + 1);
17    return min_way;
18 }
19
20 int DamLev_Recursion(string src, string trg)
21 {
22     return DamLev_RecNode(src, src.length(), trg, trg.length());
23 }

```

Листинг 3.6 — Функция рекурсивного поиска расстояния Дameraу-Левенштейна с кэшированием

```
1
2 static int DamLev_RecNodeWithCache(string src, size_t s_len, string trg, size_t
   t_len, vector<vector<int>>> &matrix)
3 {
4     if (s_len * t_len == 0)
5     {
6         int res = max(s_len, t_len);
7         matrix[s_len][t_len] = res;
8         return max(s_len, t_len);
9     }
10
11     int cost = src[s_len - 1] == trg[t_len - 1] ? 0 : 1;
12
13     int D = INT_MAX;
14     if (matrix[s_len - 1][t_len] == INT_MAX)
15     {
16         D = (DamLev_RecNodeWithCache(src, s_len - 1, trg, t_len, matrix) + 1);
17     }
18     else
19         D = matrix[s_len - 1][t_len] + 1;
20
21     int I = INT_MAX;
22     if (matrix[s_len][t_len - 1] == INT_MAX)
23     {
24         I = (DamLev_RecNodeWithCache(src, s_len, trg, t_len - 1, matrix) + 1);
25     }
26     else
27         I = matrix[s_len][t_len - 1] + 1;
28
29     int S = INT_MAX;
30     if (matrix[s_len - 1][t_len - 1] == INT_MAX)
31     {
32         S = (DamLev_RecNodeWithCache(src, s_len - 1, trg, t_len - 1, matrix) + cost);
33     }
34     else
35         S = matrix[s_len - 1][t_len - 1] + cost;
36
37     int min_way = CheckMin(D, I, S);
```


Листинг 3.7 — Функция рекурсивного поиска расстояния Дамерау-Левенштейна с кэшированием

```
1      if (s_len > 1 && t_len > 1 && src[s_len - 1] == trg[t_len - 2] && src[s_len - 2]
      == trg[t_len - 1])
2      {
3          int X = INT_MAX;
4          if (matrix[s_len - 2][t_len - 2] == INT_MAX)
5          {
6              X = DamLev_RecNodeWithCache(src, s_len - 2, trg, t_len - 2, matrix) + 1;
7          }
8          else
9              X = matrix[s_len - 2][t_len - 2] + 1;
10         min_way = min(min_way, X);
11     }
12     if (matrix[s_len][t_len] == INT_MAX || min_way < matrix[s_len][t_len])
13         matrix[s_len][t_len] = min_way;
14     return min_way;
15 }
16
17 int DamLev_RecursionWithCache(string src, string trg)
18 {
19     size_t N = src.length() + 1;
20     size_t M = trg.length() + 1;
21
22     vector<vector<int>>> matrix(N);
23
24     for (size_t i = 0; i < N; i++)
25     {
26         matrix[i].resize(M);
27         for (size_t j = 0; j < M; j++)
28             matrix[i][j] = INT_MAX;
29     }
30
31     int res = DamLev_RecNodeWithCache(src, src.length(), trg, trg.length(), matrix);
32     return res;
33 }
```

3.3 Тестирование

В таблице 3.1 отображён возможный набор тестов для тестирования методом чёрного ящика, результаты которого, представленные на рисунке 3.1, подтверждают прохождение программы перечисленных тестов.

Таблица 3.1 — Тесты проверки корректности программы

№	строка 1	строка 2	Ожидаемый результат (Л, Д-Л)	Фактический результат
1	"скат"	"скат"	0, 0	0, 0
2	"кот"	"скат"	2, 2	2, 2
3	"скат"	"скат"	0, 0	0, 0
4	"скат"	"сакт"	2, 1	2, 1

Source string: кот Target string: скат Lev Matrix: 2 Dam-Lev Matrix: 2 Dam-Lev Recursion: 2 Dam-Lev Recursion with cache: 2	Source string: скат Target string: скат Lev Matrix: 0 Dam-Lev Matrix: 0 Dam-Lev Recursion: 0 Dam-Lev Recursion with cache: 0
Source string: скат Target string: сакт Lev Matrix: 2 Dam-Lev Matrix: 1 Dam-Lev Recursion: 1 Dam-Lev Recursion with cache: 1	Source string: тест Target string: тетс Lev Matrix: 2 Dam-Lev Matrix: 1 Dam-Lev Recursion: 1 Dam-Lev Recursion with cache: 1

Рисунок 3.1 — Результаты тестирования

3.4 Сравнительный анализ потребляемой памяти

Аналитически посчитаем затрачиваемую память на строках s_1 , s_2 длиной n и m соответственно. Так как, с точки зрения памяти алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются, достаточно рассмотреть лишь разные реализации данных алгоритмов.

Использование памяти для матричного алгоритма теоритически определяется формулой (3.1):

$$V = 2(n + 1)\text{sizeof}(\text{int}) + 2\text{sizeof}(\text{int}) + 2\text{sizeof}(\text{size_t}) + \text{sizeof}(\text{char})(n + m) \quad (3.1)$$

Где:

- 1) $2(n + 1)\text{sizeof}(\text{int})$ - память под две строки (замена матрицы в моей реализации);
- 2) $2\text{sizeof}(\text{int})$ - память под размеры строк;
- 3) $2\text{sizeof}(\text{size_t})$ - память под итераторы;
- 4) $\text{sizeof}(\text{char})(n + m)$ - память под сами строки как аргументы функции.

Использование памяти для рекурсивного алгоритма зависит от максимальной глубины стека вызовов. Так как эта величина равна сумме длин входящих строк, то теоритически определяется формулой (3.2):

$$V = \text{sizeof}(\text{char})(n + m) + (n + m)(2\text{sizeof}(\text{char*}) + 3\text{sizeof}(\text{int})) \quad (3.2)$$

Где:

- 1) $\text{sizeof}(\text{char})(n + m)$ - память под аргументы функции;
- 2) $(n + m)(2\text{sizeof}(\text{char}*) + 3\text{sizeof}(\text{int}))$ - выделенные рекурсией ресурсы умноженные на максимальную глубину стека вызовов.

Вывод

Были разработаны и протестированы спроектированные алгоритмы, приведены расчеты сравнительного анализа потребляемой памяти, а также указаны средства реализации поставленной задачи.

4 Экспериментальный раздел

В данном разделе будут проведены эксперименты для проведения сравнительного анализа алгоритмов по затрачиваемому процессорному времени.

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

В рамках данного проекта были проведёны следующие эксперименты:

1) сравнение нерекурсивных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна на строках длиной от 5 до 10 с шагом 1 (график 4.1);

2) сравнение рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна и рекурсивного алгоритма Дамерау-Левенштейна с кэшированием на строках длиной от 0 до 12 с шагом 1 (график 4.2).

Тестирование проводилось на компьютере с процессором Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz 1.80 GHz под управлением Windows 11 с 8 Гб оперативной памяти.

Ниже представлены результаты замеров (таблица 4.1) для всех 4 алгоритмов ¹.

Таблица 4.1 — Замеры времени (в секундах) для строк различной длины

Длина строк	Л. матричный	Д.Л. матричный	Д.Л. рекурсивный	Д.Л. с кэшем
5	2.5e-0.5	0.002	5.3e-05	0.003
6	3.4e-0.5	0.009	6.7e-05	0.015
7	4.3e-0.5	0.031	8.5e-05	0.046
8	5.3e-0.5	0.203	0.000101	0.203
9	7e-0.5-	1.093	0.00013	1.265
10	9.2-0.5	5.937	0.00015	6.625
20	0.0003	-	0.0006	-
40	0.001	-	0.002	-
60	0.002	-	0.004	-
80	0.004	-	0.007	-
100	0.007	-	0.012	-
120	0.0108	-	0.018	-
140	0.015	-	0.024	-
160	0.019	-	0.037	-

¹Прочерк в таблице означает, что для заданных значений тестирование не проводилось

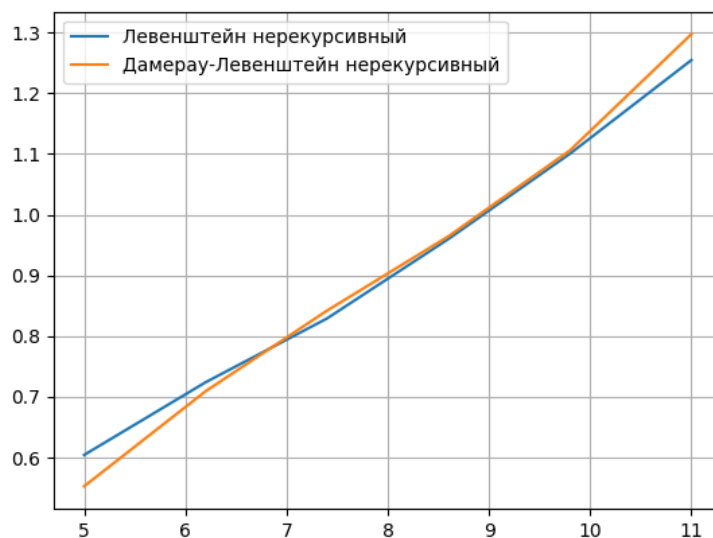


Рисунок 4.1 — Зависимость времени работы алгоритмов от длин строк



Рисунок 4.2 — Зависимость времени работы алгоритмов от длин строк

4.2 Вывод

В данном разделе были поставлены эксперименты по замеру времени выполнения каждого из алгоритмов. Рекурсивные алгоритмы по нахождению редакционного расстояния работают дольше чем матричные на несколько порядков. Время работы таких реализаций увеличивается в геометрической прогрессии. С этой проблемой помогает справляться кэширование, но такой подход увеличивает затраты в памяти.

Следовательно, матричная реализация более применима в реальных проектах.

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).

Цель работы была достигнута.

Выполненные задачи:

- 1) выбраны инструменты для замера процессорного времени выполнения реализации алгоритмов;
- 2) изучены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна;
- 3) реализованы:
 - а) нерекурсивный метод поиска расстояния Левенштейна;
 - б) нерекурсивный метод поиска расстояния Дамерау-Левенштейна;
 - в) рекурсивный метод поиска расстояния Дамерау-Левенштейна;
 - г) рекурсивный с кэшированием метод поиска расстояния Дамерау-Левенштейна;
- 4) замерены процессорное время и потребленную память для всех реализованных алгоритмов;
- 5) проведены анализ работы программы по времени и по памяти, выяснить влияющие на них характеристики.

Список источников

- 1) Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. - М.: Доклады АН СССР, 1965. Т.163. С.845-848.
- 2) Нечеткий поиск в словаре с универсальным автоматом Левенштейна. Режим доступа: <https://habr.com/ru/post/275937/> (дата обращения: 10.09.2021).
- 3) Нечеткий поиск в тексте и словаре. Режим доступа: <https://habr.com/ru/post/114997/> (дата обращения: 11.09.2021).
- 4) Functools — Higher-order functions and operations on callable objects. Режим доступа: <https://docs.python.org/3/library/functools.html> (дата обращения: 11.09.2021).
- 5) Time — Time access and conversions Режим доступа: <https://docs.python.org/3/library/time.html> (дата обращения: 11.09.2021).