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»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

по курсу «Анализ алгоритмов»

на тему:

«Редакционные расстояния»

Студент Рунов К. А.

Группа ИУ7-54Б

Преподаватели Волкова Л. Л., Строганов Д. В.

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Редакционные расстояния — расстояние Левенштейна и его модификация — расстояние Дамерау — Левенштейна — метрики сходства между двумя символьными последовательностями. Расстоянием в этих метриках считается минимальное число односимвольных преобразований (удаления, вставки, замены или транспозиции), необходимых для преобразования одной последовательности символов в другую.

Редакционные расстояния применяются, например, в следующих ситуациях:

— для исправления ошибок в слове поискового запроса;

— в формах заполнения информации на сайтах;

— для распознавания рукописных символов;

— в базах данных [1].

Целью данной лабораторной работы является реализация и исследование алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна.

Для достижения поставленной цели нужно решить следующие задачи:

1) описать алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна;

2) обосновать выбор средств реализации алгоритмов;

3) реализовать итерационный алгоритм нахождения расстояния Левенштейна;

4) реализовать итерационный алгоритм нахождения расстояния Дамерау — Левенштейна;

5) реализовать рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна;

6) реализовать рекурсивный с кешированием алгоритм нахождения расстояния Дамерау — Левенштейна;

7) провести сравнительный анализ реализаций алгоритмов по используемому процессорному времени и максимальной затрачиваемой памяти;

8) описать и проанализировать полученные результаты в отчёте.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна — метрика, определяющая понятие расстояния между двумя последовательностями символов, как минимального количества редакторских операций вставки (I, от англ. insert), замены (R, от англ. replace) и удаления (D, от англ. delete), необходимых для преобразования одной строки в другую [2]. Для каждой операции должна быть определена её стоимость. Введём обозначения для стоимостей. Пусть:

1) $w(a, b)$ — цена замены символа a на b;

2) $w(\lambda, b)$ — цена вставки символа b;

3) $w(a, \lambda)$ — цена удаления символа a.

Определим стоимости операций:

$$w(a, b) = \begin{cases} 1, & \text{если } a \neq b; \\ 0, & \text{иначе.} \end{cases} \tag{1}$$

Отсутствие операций в случае совпадения символов будем обозначать за M (от англ. match).

Введём в рассмотрение функцию $D(S_1[1..i], S_2[1..j])$, значнием которой является редакционное расстояние между подстроками $S_1[1..i]$ и $S_2[1..j]$, где $S_1[1..i]$ — подстрока S_1 длины i. Так, если S_1 = "скат", то $S_1[1..0] = \lambda$, $S_1[1..1]$ = "с", $S_1[1..2]$ = "ск". Расстояние Левенштейна между подстроками S_1 и S_2 вычисляется по рекуррентной формуле:

$$D(S_1[1..i], S_2[1..j]) = \\ = \begin{cases} \max(i,\ j), & i \cdot j = 0; \\ \min(D(S_1[1..i], S_2[1..j-1]) + 1, & \\ \qquad D(S_1[1..i-1], S_2[1..j]) + 1, & i \cdot j \neq 0. \\ \qquad D(S_1[1..i-1], S_2[1..j-1]) + w(S_1[i], S_2[j])), & \end{cases} \tag{2}$$

Итерационный алгоритм нахождения расстояния Левенштейна

Рекурсивная реализация алгоритма поиска расстояния Левенштейна малоэффективна по времени при больших L_1 и L_2, так как производится много повторных, лишних вычислений. Реализацию можно оптимизировать с помощью динамического программирования. Например, ввести матрицу размерности $(L_1 + 1) \times (L_2 + 1)$ и заполнять её промежуточными значениями $D(S_1[1..i], S_2[1..j])$, используя их затем по ходу вычислений. Значения в ячейках $[i][j]$ (i-я строка, j-й столбец) матрицы равны значениям $D(S_1[1..i], S_2[1..j])$ соответственно. Можно заметить, что всю матрицу для вычислений хранить не обязательно — двух строк будет достаточно.

1.2 Расстояние Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна — метрика, которая определяет расстояние между двумя последовательностями символов, как и расстояние Левенштейна, но к исходному набору редакторских операций добавляется ещё одна — транспозиция (T, от англ. transposition). Операция транспозиции меняет местами соседние символы в строке. Обозначим её стоимость: $w(ab, ba) = 1$.

Расстояние Дамерау — Левенштейна $\mathcal{D}(S_1, S_2)$ между подстроками S_1 и S_2 соответственно может быть вычислено по формуле:

$$
\begin{aligned}
&\mathcal{D}(S_1[1..i], S_2[1..j]) = \\
&= \begin{cases}
\max(i,\ j), & i \cdot j = 0; \\
\min\Big(\mathcal{D}(S_1[1..i], S_2[1..j-1]) + 1, & \\
\qquad \mathcal{D}(S_1[1..i-1], S_2[1..j]) + 1, & \\
\qquad \mathcal{D}(S_1[1..i-1], S_2[1..j-1]) + w(S_1[i], S_2[j]), & \\
\qquad \begin{cases}
 & i > 1,\ j > 1, \\
\mathcal{D}(S_1[1..i-2], S_2[1..j-2]) + 1, & S_1[i] = S_2[j-1], \\
 & S_1[i-1] = S_2[j]; \\
\infty, & \text{иначе},
\end{cases}\Big) & i \cdot j \neq 0.
\end{cases}
\end{aligned}
\tag{3}
$$

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна реализует рекуррентную формулу (3). Таким образом, верно следующее:

1) $\mathcal{D}(\lambda, \lambda) = 0$, — для преобразования пустой строки в пустую строку требуется 0 операций вставки, замены, удаления и транспозиции.

2) $\mathcal{D}(S_1, \lambda) = |S_1|$ (длина S_1), — для преобразования строки S_1 в пустую строку требуется $|S_1|$ операций (удаления).

3) $\mathcal{D}(\lambda, S_2) = |S_2|$, — для преобразования пустой строки в строку S_2 требуется $|S_2|$ операций (вставки).

4) $\mathcal{D}(c_1, c_2) = \begin{cases} 1, \ c_1 \neq c_2; \\ 0, \ c_1 = c_2, \end{cases}$ — для преобразования одного символа в другой требуется 1 операция (замены), если символы отличаются, и 0 операций, если символы совпадают.

5) Для преобразования одной пары символов в другую,
$$\mathcal{D}(c_1c_2, c_3c_4) = \begin{cases} 0, \ c_1 = c_3, \ c_2 = c_4; \ // \ MM \\ 1, \ c_1 = c_3, \ c_2 \neq c_4; \ // \ MR \\ 1, \ c_1 \neq c_3, \ c_2 = c_4; \ // \ RM \\ 1, \ c_1 = c_4, \ c_2 = c_3; \ // \ T \\ 2, \ \text{иначе.} \ // \ RR \end{cases}$$

6) Пусть $S_1 = S_1'c_2 = S_1''c_1c_2$, $S_2 = S_2'c_4 = S_2''c_3c_4$, где S_1' и S_2' — строки S_1 и S_2 без последних символов, S_1'' и S_2'' — строки S_1 и S_2 без двух последних символов, а c_1c_2 и c_3c_4 — пары их последних символов соответственно.

$$\mathcal{D}(S_1, S_2) = \min \begin{cases} \mathcal{D}(S_1'c_2, S_2') + 1, \\ \mathcal{D}(S_1', S_2'c_4) + 1, \\ \mathcal{D}(S_1', S_2') + w(c_2, c_4), \\ \mathcal{D}(S_1'', S_2'') + 1, & \text{если } c_2 = c_3, \ c_1 = c_4. \end{cases}$$

Можно заметить, что $\mathcal{D}(S_1, S_2)$ вычисляется как минимальная длина последовательности редакторских операций, которыми можно преобразовать строку S_1 в S'_2 плюс цена вставки последнего символа из S_2, строку S'_1 в S_2 плюс цена удаления последнего символа из S_1, строку S'_1 в S'_2 плюс цена замены последних символов строк S_1 и S_2, строку S''_1 в S''_2 плюс цена транспозиции двух последних символов строк, если она возможна. Для любых подстрок $\mathcal{S}_1$ и $\mathcal{S}_2$ строк S_1 и S_2, $\mathcal{D}(\mathcal{S}_1, \mathcal{S}_2)$ вычисляет минимальное количество редакторских операций. Следовательно, $\mathcal{D}(S_1, S_2)$ действительно считает расстояние Дамерау — Левенштейна для произвольных строк строк S_1 и S_2.

1.2.2 Рекурсивный с кешированием алгоритм нахождения расстояния Дамерау — Левенштейна

Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна прост, но его реализация на ЭВМ без дополнительных оптимизаций нерациональна, так как по нескольку раз считаются значения, которые уже были вычислены, а вычисление их может оказаться достаточно трудоёмким.

Идея возможной оптимизации состоит в следующем: хранить получаемые по ходу выполнения алгоритма значения в матрице, а перед вычислением очередного — проверять, было ли оно посчитано ранее (заполнена ли соответствующая ячейка матрицы), и, если да, — брать его оттуда, не прибегая к повторным вычислениям. Для того, чтобы узнать, заполнена ячейка матрицы или нет, нужно изначально проинициализировать все ячейки матрицы какими-то значениями, которые на практике не могут быть получены в результате рассчёта расстояния Дамерау — Левенштейна, например, -1.

1.2.3 Итерационный алгоритм нахождения расстояния Дамерау — Левенштейна

Как алгоритм поиска расстояния Левенштейна, так и алгоритм поиска расстояния Дамерау — Левенштейна можно реализовать нерекурсивно с помощью динамического программирования, используя матрицу расстояний.

Процесс вычисления значения ячейки матрицы показан на рисунке 1.1.

$$
\begin{array}{c|c|c|c|}
 & s_{21} & s_{22} & s_{23} \\
\hline
s_{11} & N_T & \cdots & \cdots \\
\hline
s_{12} & \vdots & N_R & N_D \\
\hline
s_{13} & \vdots & N_I & \boldsymbol{N} \\
\hline
\end{array}
$$

$$
\boldsymbol{N} = \min \begin{cases} N_I + 1, \\ N_D + 1, \\ N_R + w(s_{13}, s_{23}), \\ \begin{cases} N_T + 1, & \text{если } s_{13} = s_{22},\ s_{12} = s_{23} \text{ и } s_{11},\ s_{21} \text{ существуют,} \\ \infty, & \text{иначе.} \end{cases} \end{cases}
$$

Рисунок 1.1 – Вычисление расстояния Дамерау — Левенштейна с использованием матрицы

Таким образом, для нахождения расстояния Дамерау — Левенштейна хранить всю матрицу не обязательно — трёх строк будет достаточно. Заполнив последнее значение очередной строки, можно выполнить «циклическую прокрутку» строк матрицы вверх, после чего изменить значение первой ячейки последней теперь строки матрицы на значение первой ячейки предпоследней строки матрицы, увеличенное на единицу, а затем продолжить заполнение матрицы по алгоритму, представленному на рисунке выше, перезаписывая значения ячеек.

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна. Поскольку данные расстояния могут быть вычислены с помощью рекуррентных формул, то алгоритмы могут быть реализованы как рекурсивно, так и итерационно.

2 Конструкторская часть

В этой части будут приведены требования к программе и схемы алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна.

2.1 Требования к программному обеспечению

Программа должна поддерживать два режима работы: ручной ввод и массированный замер времени.

В режиме ручного ввода пользователь должен иметь возможности

— ввести два слова через пробел с клавиатуры;

— использовать символы в кодировке Unicode;

— ввести слова размером до 25 000 символов в кодировке Unicode.

В режиме ручного ввода программа должна выполнять следующие действия:

— запустить каждую из реализаций четырёх алгоритмов (итерационного алгоритма нахождения расстояния Левенштейна, итерационного алгоритма нахождения расстояния Дамерау — Левенштейна, рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна, рекурсивного с кешированием алгоритма нахождения расстояния Дамерау — Левенштейна);

— вывести заполненные матрицы расстояний для итерационных алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна;

— вывести последовательность редакторских операций, которая привела к нахождению расстояний Левенштейна и Дамерау — Левенштейна, для итерационных алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна;

— вывести результат работы каждого алгоритма с названием этого алгоритма.

В режиме массированного замера времени пользователь должен иметь следующие возможности:

— ввести начальную длину строк, конечную длину строк, шаг изменения длин строк, которые будут использоваться в замерах времени;

— ввести количество замеров времени, которое требуется произвести для каждой длины строк (в результате будет выдано среднее значение по каждой длине);

— посмотреть графики, построенные на основе данных из последнего замера времени.

В режиме массированного замера времени программа должна выполнять следующие действия:

— на основе данных о длинах строк, полученных от пользователя, сгенерировать строки указанных длин из фиксированного набора символов в кодировке Unicode; конкретный символ строки должен выбираться в результате генерации псевдослучайного целого числа по модулю количества символов в фиксированном наборе — полученное число есть индекс символа в наборе;

— каждую пару сгенерированных строк подать на вход всем алгоритмам, время работы реализаций которых требуется измерить;

— произвести замеры времени работы для каждой реализации алгоритмов без учёта времени на создание и инициализацию матрицы в тех алгоритмах, где матрица используется;

— записать данные о каждом замере времени в файл;

— на основе данных из файла построить четыре графика (по графику на каждый алгоритм), где значения по оси Ox — длины строк, значения по оси Oy — время работы реализации алгоритма в наносекундах, и вывести результат пользователю.

2.2 Разработка алгоритмов

На вход алгоритмам подаются строки $S1$, $S2$ и их длины $L1$, $L2$ соответственно.

На рисунке 2.1 представлена схема алгоритма нахождения расстояния Левенштейна. На рисунках 2.2 – 2.5 представлены схемы алгоритмов нахождения расстояния Дамерау — Левенштейна.

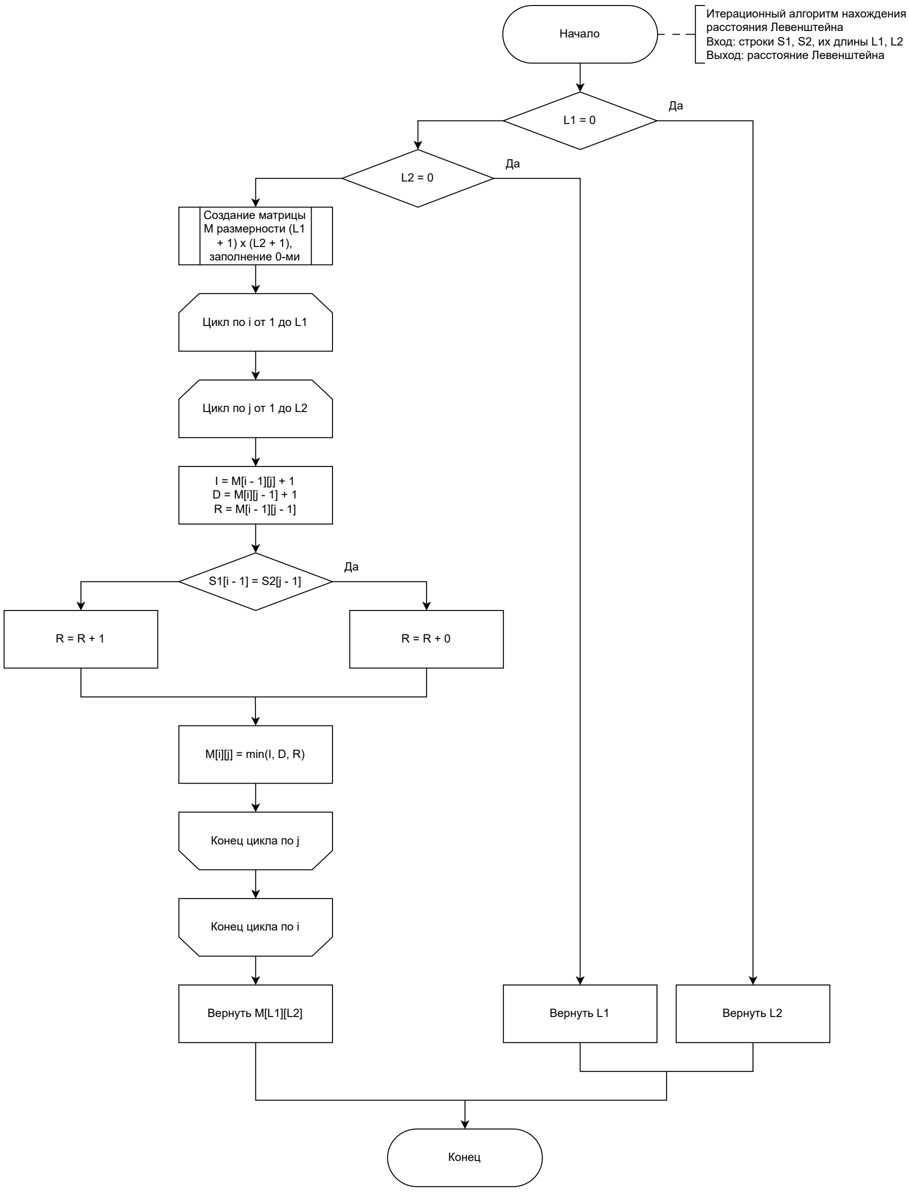

Рисунок 2.1 – Итерационный алгоритм нахождения расстояния Левенштейна

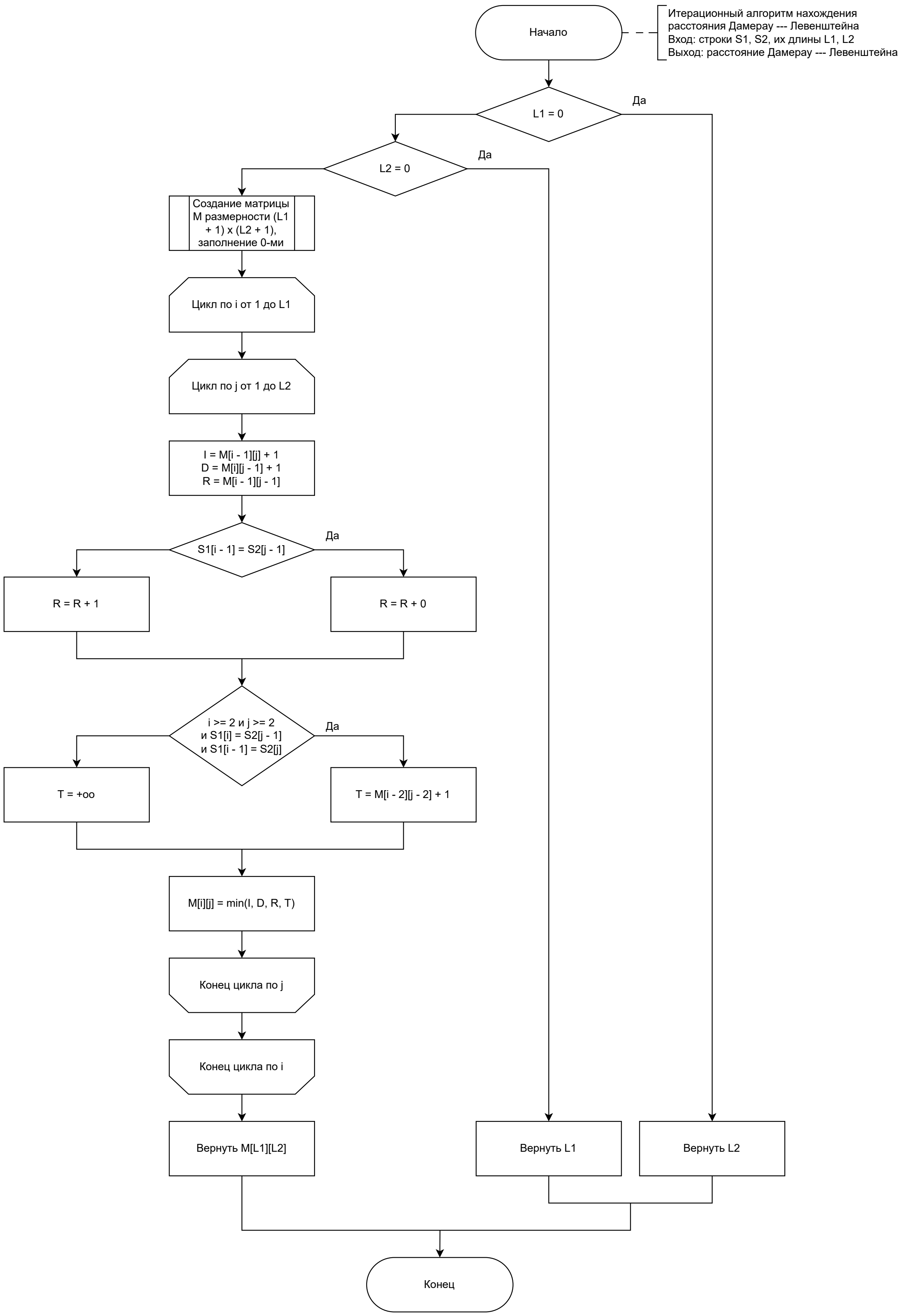

Рисунок 2.2 – Итерационный алгоритм нахождения расстояния Дамерау — Левенштейна

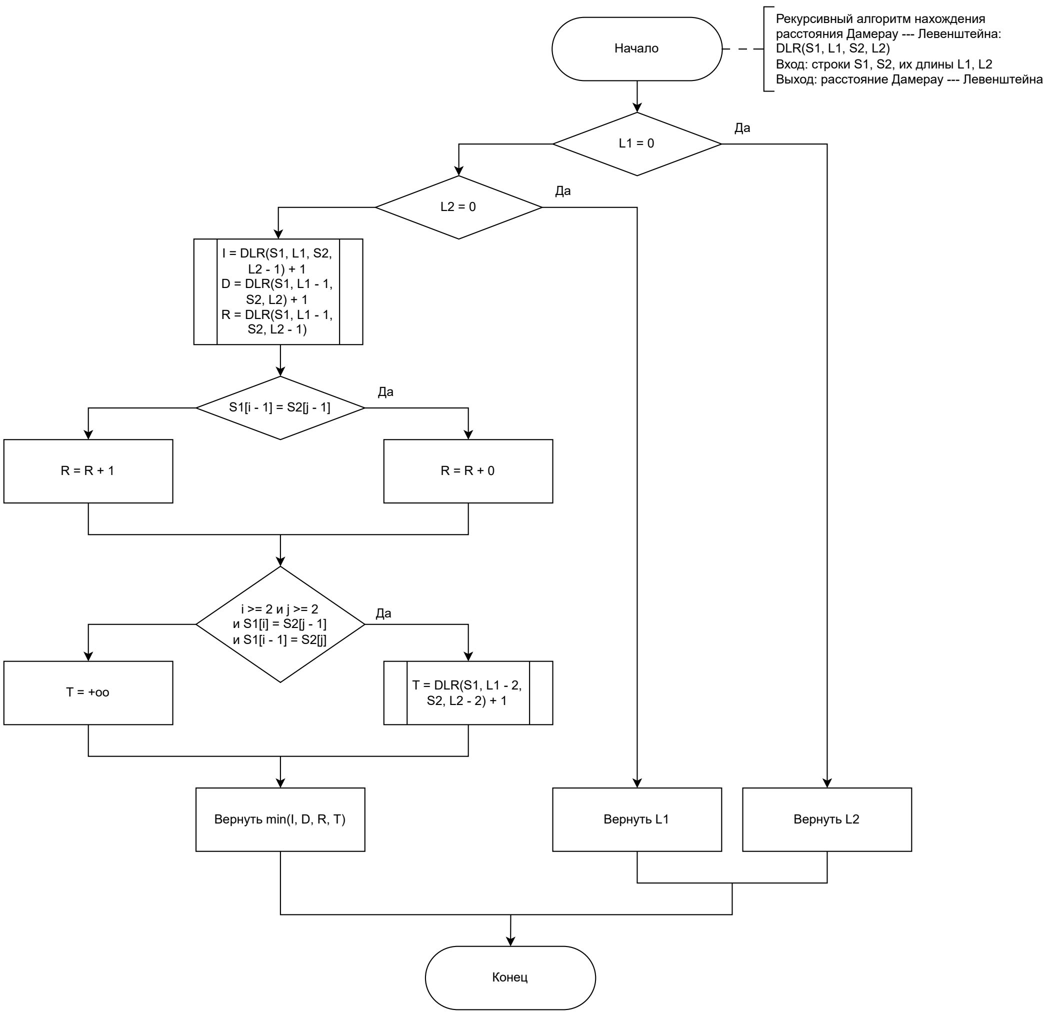

Рисунок 2.3 – Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

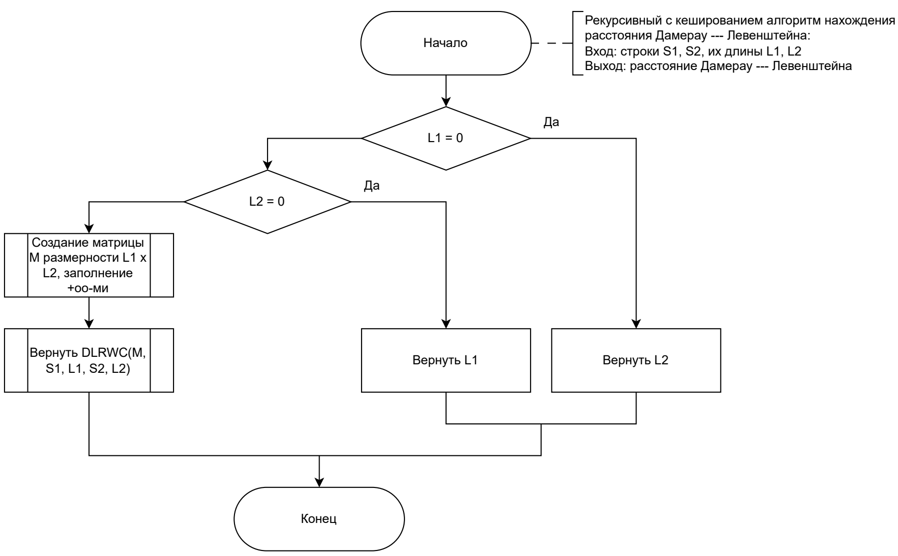

Рисунок 2.4 – Рекурсивный с кешированием алгоритм нахождения расстояния Дамерау — Левенштейна

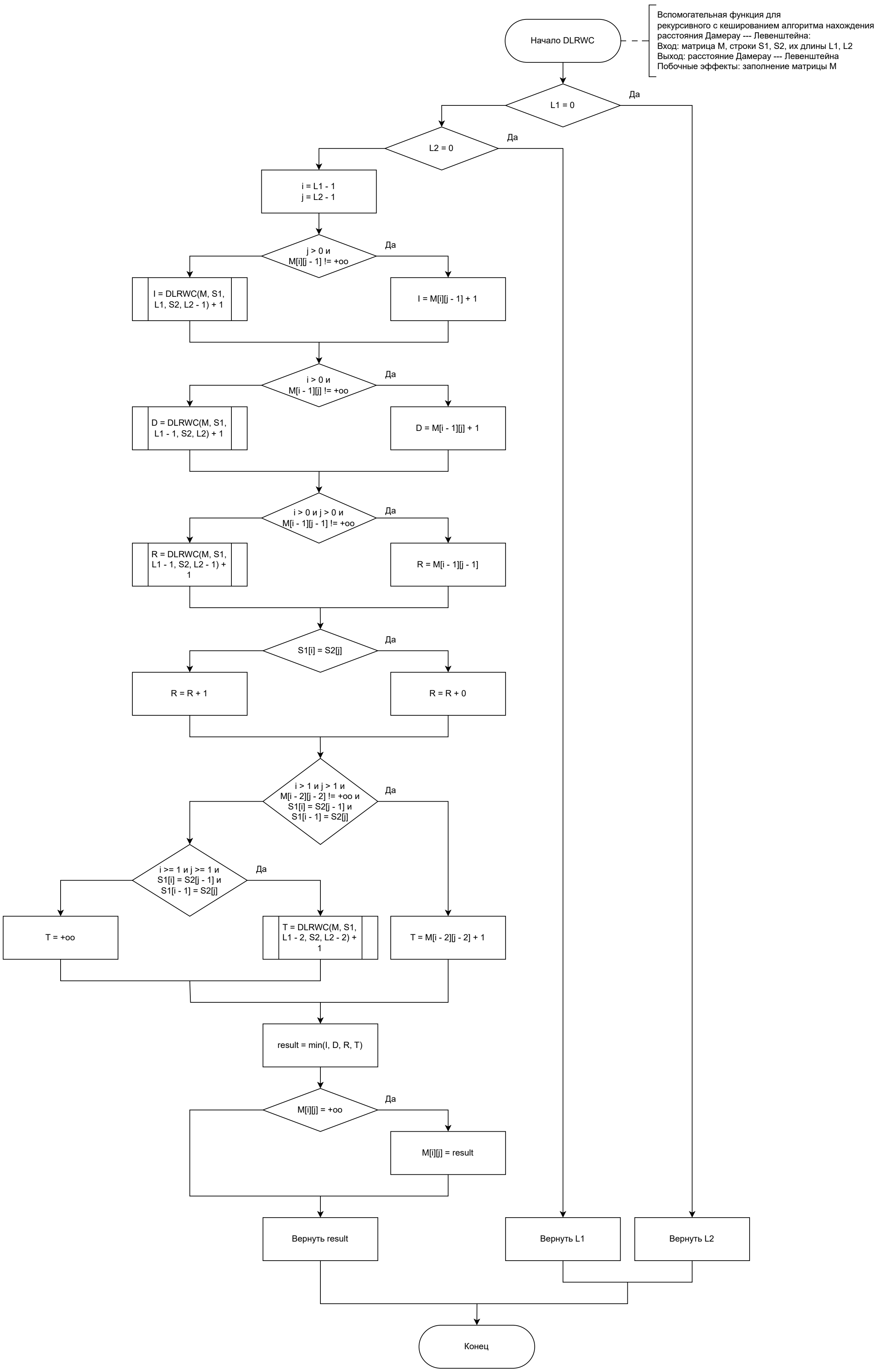

Рисунок 2.5 – Рекурсивная функция нахождения расстояния Дамерау — Левенштейна с использованием матрицы в качестве кеша

2.3 Описание типов и структур данных

Для реализации алгоритмов будут использованы следующие типы и структуры данных:

— wchar_t — для символов в кодировке Unicode, из которых состоят строки;

— wchar_t* — для строк, подаваемых на вход;

— size_t (unsigned long int) — для длин строк; для вычисленных расстояний Левенштейна и Дамерау — Левенштейна (как возвращаемых из функций, так и хранящихся в матрице); для размеров матрицы при её создании;

— size_t* — для строк матрицы;

— size_t ** — для матрицы (массива указателей на строки);

— struct BenchmarkData (см. листинг 1) — для временного хранения данных о замерах времени выполнения реализаций алгоритмов.

Листинг 1 – Структура для временного хранения данных о замерах времени выполнения реализаций алгоритмов

```
struct BenchmarkData
{
    size_t string_length;
    long t_lev_iter_ns;
    long t_damlev_iter_ns;
    long t_damlev_rec_ns;
    long t_damlev_rec_cache_ns;
};
```

Вывод

В данном разделе на основе теоретических данных были построены схемы алгоритмов, выбраны типы и структуры данных, которые будут использоваться при реализации алгоритмов.

3 Технологическая часть

В данном разделе будут описаны требования к программному обеспечению, средства реализации и функциональные тесты. Также будут приложены листинги кода.

3.1 Средства реализации

Для реализации данной работы был выбран язык C++ [3] по следующим причинам:

— имеется опыт разработки на данном языке;

— в стандартной библиотеке языка присутствует функция clock_gettime [4], которая позволяет рассчитать процессорное время выполнения конкретного потока;

— наличие библиотеки для проведения автоматизированного тестирования gtest [5];

— язык поддерживается отладчиком gdb [6].

Для построения графиков был выбран язык Python [7], так как в нём есть все необходимые для этого библиотеки [8] [9] [10].

В качестве среды разработки был выбран Neovim [11].

3.2 Сведения о модулях программы

Программа поделена на следующие модули:

— main.cpp — файл, содержащий точку входа в программу, из которой происходит вызов алгоритмов по разработанному интерфейсу;

— func.cpp — файл, содержащий реализации алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна;

— test.cpp — файл, содержащий модульные тесты и точку входа в программу для их запуска;

— benchmark.cpp — файл, содержащий функции для проведения замеров времени работы реализаций алгоритмов;

— globals.cpp — файл, содержащий глобальные переменные для замеров количества рекурсивных вызовов и максимального размера стека;

— plot.py — файл, содержащий программу для построения графиков по данным о времени выполнения реализаций алгоритмов.

3.3 Реализация алгоритмов

Листинги исходных кодов программ 2 – 12 приведены в приложении.

3.4 Функциональные тесты

В таблице 1 приведены функциональные тесты для алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна. Все тесты пройдены успешно.

Таблица 1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм нахождения расстояния**			
Строка 1	**Строка 2**	**Левенштейна**	**Дамерау — Левенштейна**		
		Итерационный	**Итерационный**	**Рекурсивный**	
				Без кеша	**С кешем**
λ	λ	0	0	0	0
abc	λ	3	3	3	3
λ	a	1	1	1	1
a	a	0	0	0	0
a	b	1	1	1	1
$\xi\eta$	$\eta\xi$	2	1	1	1
$abcab$	$bacba$	4	2	2	2
$abcabcab$	$bacbacba$	6	3	3	3
$python$	$ptyhon$	2	1	1	1
скат	кот	3	2	2	2
ls	sl	2	1	1	1
$make$	$mkae$	2	1	1	1

Вывод

Были реализованы и протестированы реализации следующих алгоритмов:

— Итерационный алгоритм нахождения расстояния Левенштейна;

— Итерационный алгоритм нахождения расстояния Дамерау — Левенштейна;

— Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна;

— Рекурсивный с кешированием алгоритм нахождения расстояния Дамерау — Левенштейна.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программ и сравнительный анализ алгоритмов на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени, следующие

— Процессор: AMD Ryzen™ 7 4700U 2.0 ГГц [12], 8 физических ядер, 8 потоков;

— Оперативная память: 8 ГБ, DDR4, 3200 МГц;

— Операционная система: NixOS 23.05.4448.5550a [13];

— Версия ядра: 6.1.59.

4.2 Демонстрация работы программы

На рисунках 4.1 – 4.3 представлена демонстрация работы программы: ручной ввод слов, проведение замеров времени и построение графиков.

```
human in ~/University/aa on main ● ● λ make run
cd build && ./main
Выберите действие:
  1 – Ввести два слова
  2 – Провести замеры времени
  3 – Провести замеры времени (ручной ввод)
  4 – Построить графики
  5 – Выйти
> 1
abcab bacba
Левенштейн итерационный:
     λ  b  a  c  b  a
 λ   0  1  2  3  4  5
 a   1  1  1  2  3  4
 b   2  1  2  2  2  3
 c   3  2  2  2  3  3
 a   4  3  2  3  3  3
 b   5  4  3  3  3  4
Редакторские операции: RRMRR
Результат: 4

Дамерау–Левенштейн итерационный:
     λ  b  a  c  b  a
 λ   0  1  2  3  4  5
 a   1  1  1  2  3  4
 b   2  1  1  2  2  3
 c   3  2  2  1  2  3
 a   4  3  2  2  2  2
 b   5  4  3  3  2  2
Редакторские операции: TMT
Результат: 2

Левенштейн Итерационный                      : 4
Дамерау–Левенштейн Итерационный              : 2
Дамерау–Левенштейн Рекурсивный               : 2
Дамерау–Левенштейн Рекурсивный с кешированием : 2
```

Рисунок 4.1 – Демонстрация работы программы, ввод двух слов

```
human in ~/University/aa on main ● ● λ make run
cd build && ./main
Выберите действие:
  1 – Ввести два слова
  2 – Провести замеры времени
  3 – Провести замеры времени (ручной ввод)
  4 – Построить графики
  5 – Выйти
> 3
Введите начальную длину слов, конечную, шаг изменения
 длины, и сколько замеров времени требуется провести
для каждой длины:
1 10 1 10
Производится замер времени...
Длина слов = 1...
Длина слов = 2...
Длина слов = 3...
Длина слов = 4...
Длина слов = 5...
Длина слов = 6...
Длина слов = 7...
Длина слов = 8...
Длина слов = 9...
Длина слов = 10...
Данные записаны в файл.
```

Рисунок 4.2 – Демонстрация работы программы, проведение замеров времени

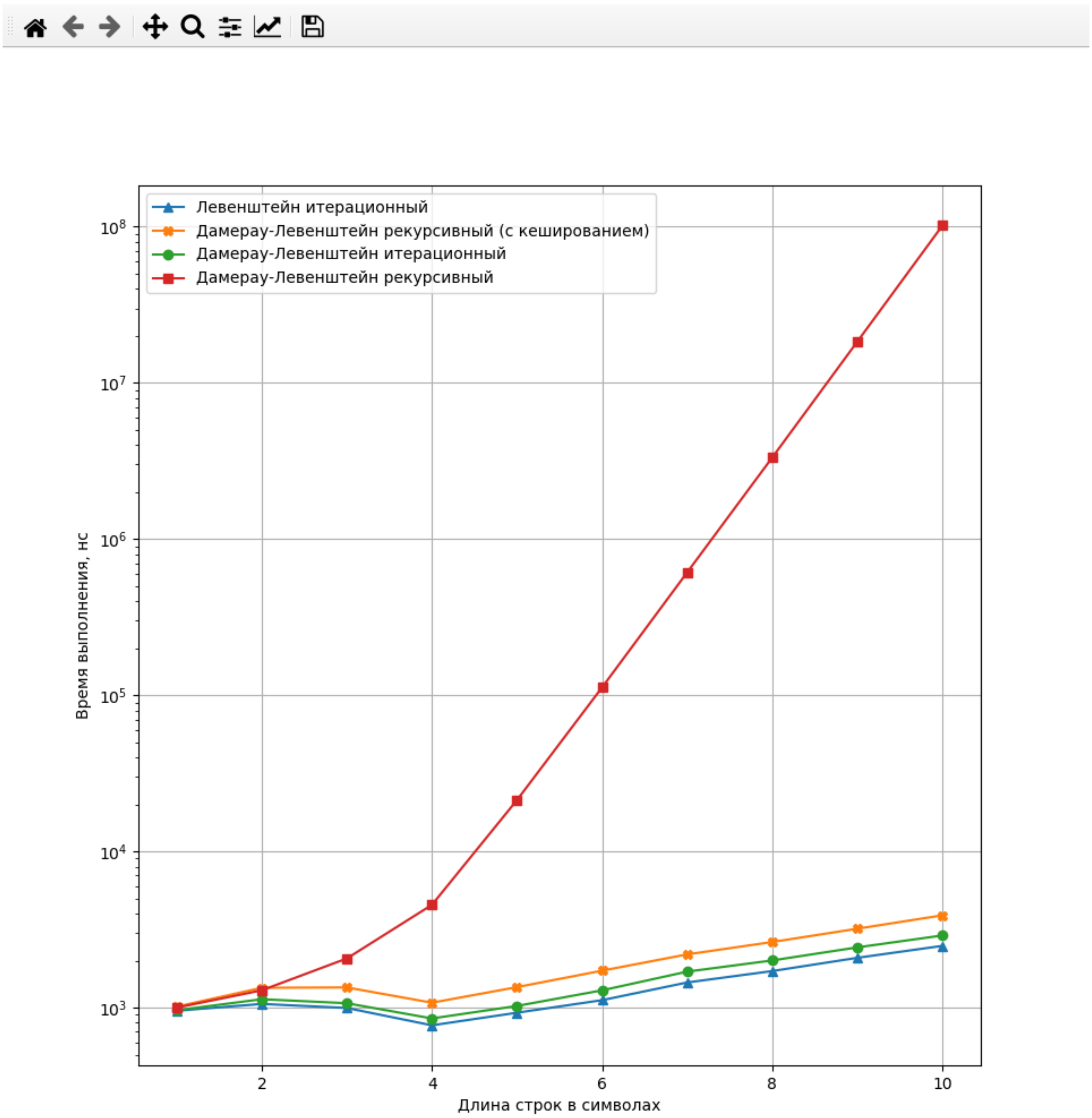

Рисунок 4.3 – Демонстрация работы программы, построение графиков на основе последних замеров времени (создаётся дочерний процесс, который запускает Python-скрипт)

4.3 Временные характеристики

На рисунках 4.4 – 4.8 представлены результаты замеров времени выполнения реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна. Замеры времени проводились для строк одинаковой длины. Отображённое на графиках время является усреднённым для каждой длины строк.

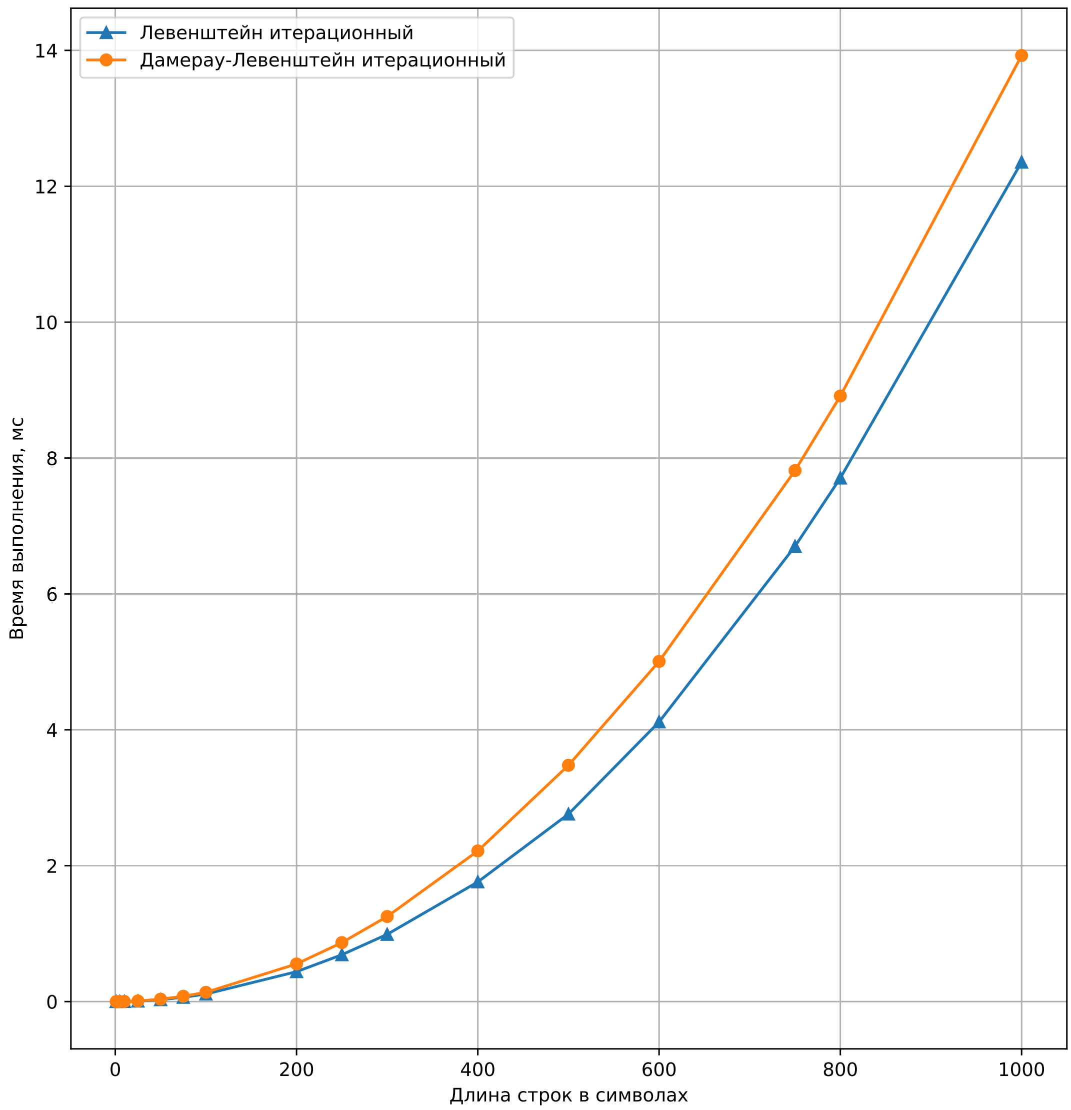

Рисунок 4.4 – Сравнение реализаций итерационных алгоритмов по времени выполнения (среднее время из 100 замеров)

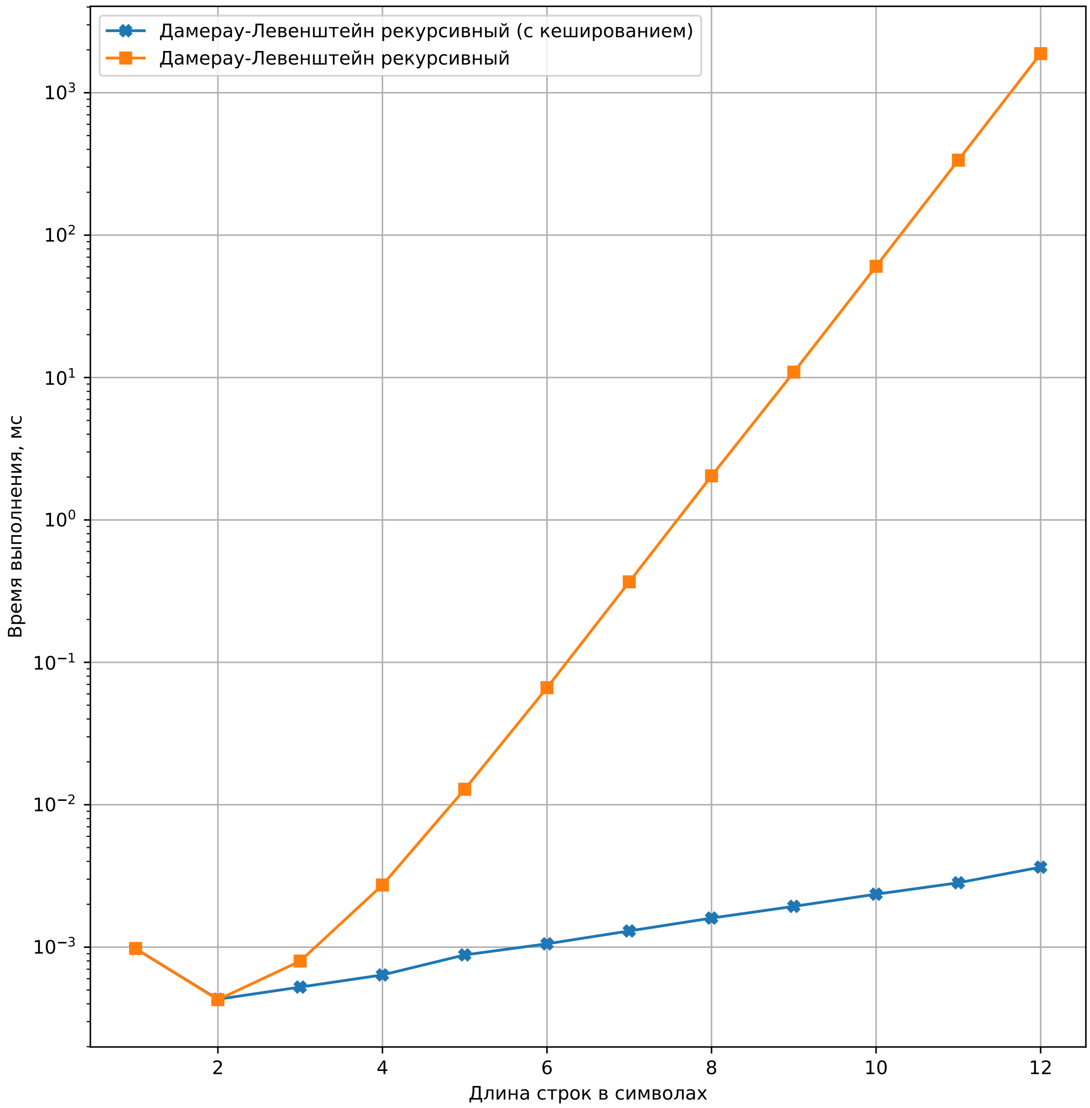

Рисунок 4.5 – Сравнение реализаций рекурсивных алгоритмов по времени выполнения (среднее время из 10 замеров)

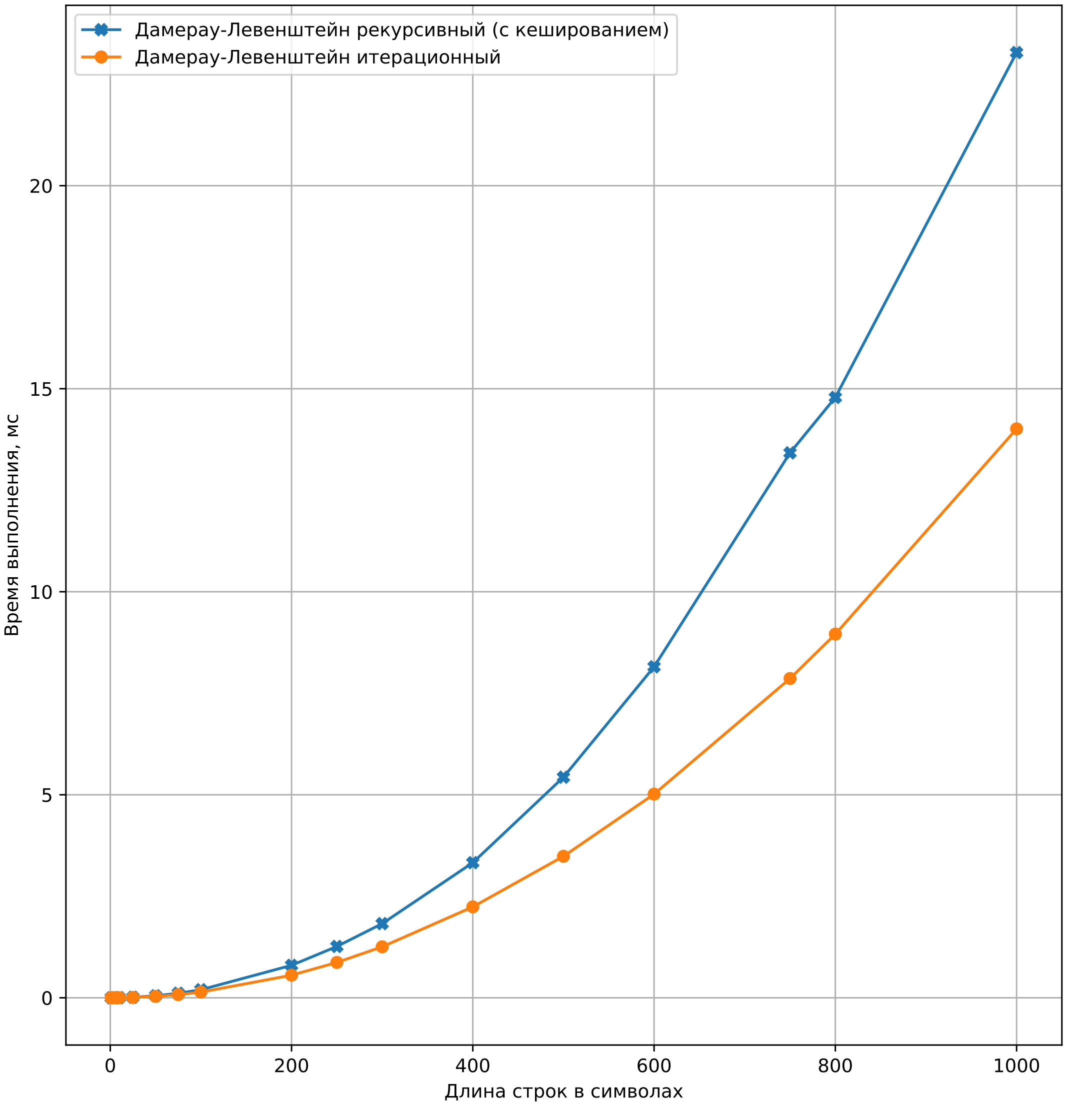

Рисунок 4.6 – Сравнение итерационной и рекурсивной с кешированием реализации алгоритмов нахождения расстояния Дамерау — Левенштейна по времени выполнения (среднее время из 100 замеров)

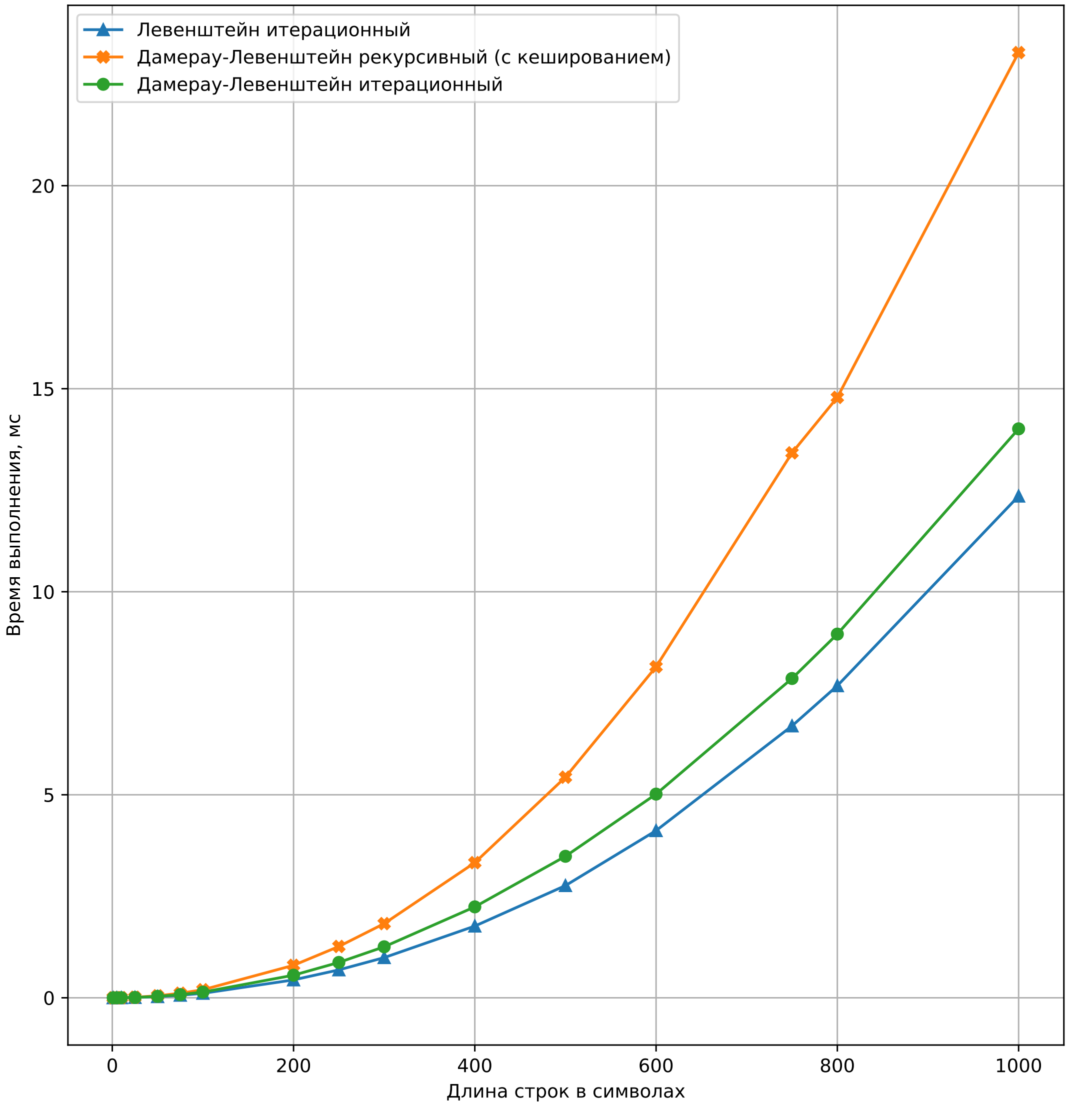

Рисунок 4.7 – Сравнение итерационных и рекурсивной с кешированием реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна по времени выполнения (среднее время из 100 замеров)

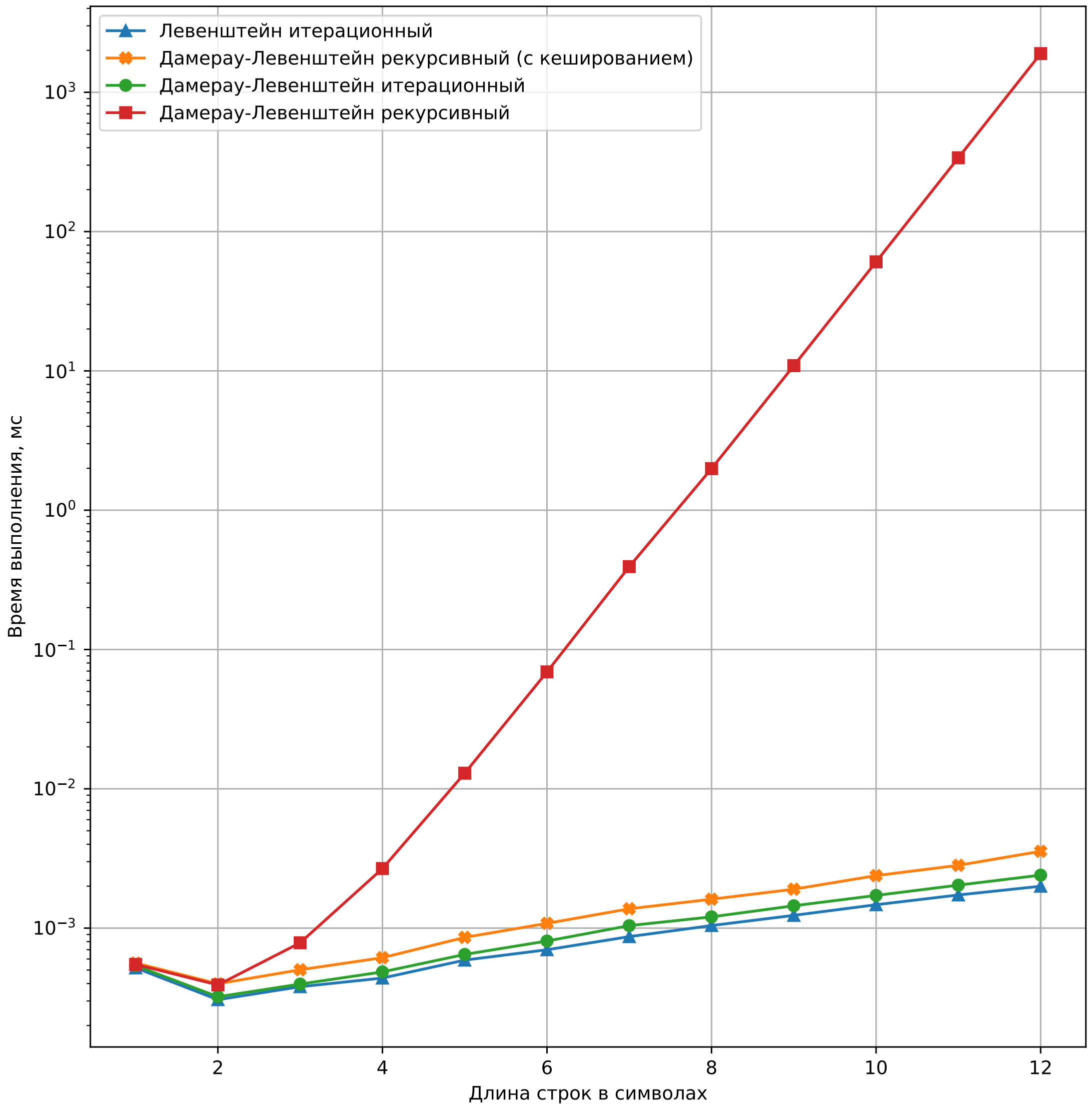

Рисунок 4.8 – Сравнение всех реализаций алгоритмов по времени выполнения (среднее время из 10 замеров)

4.4 Теоретические оценки затрачиваемой памяти

Введём следующие обозначения:

— L_1 — длина первой строки;

— L_2 — длина второй строки;

— $char$ — символьный тип данных;

— $*$ — тип данных — указатель;

— int — целочисленный тип данных;

— $size()$ — функция, вычисляющая размер типа данных в байтах, т.е. на практике $size(char) \sim sizeof(wchar\_t)$;

На основе введённых обозначений проведём теоретическую оценку памяти, используемой реализациями алгоритмов.

4.4.1 Оценка памяти реализаций итерационных алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна

Далее представлены типы данных и память, которую они занимают:

— Первая строка: $L_1 \cdot size(char)$;

— Вторая строка: $L_2 \cdot size(char)$;

— Размеры строк: $2 \cdot size(int)$;

— Указатели на строки и адрес возврата: $3 \cdot size(*)$;

— Матрица: $(L_1 + 1) \cdot (L_2 + 1) \cdot size(int) + (L_1 + 1) \cdot size(*) + size(*)$;

— Вспомогательные переменные: $c \cdot size(int)$, где c — количество вспомогательных переменных;

Итого, функция суммарного объёма используемой памяти, имеет вид:

$$\begin{aligned} V(L_1, L_2) = L_1 \cdot size(char) + L_2 \cdot size(char) \\ + 2 \cdot size(int) + (L_1 + 1) \cdot (L_2 + 1) \cdot size(int) \\ + 3 \cdot size(*) + (L_1 + 1) \cdot size(*) + size(*) + c \cdot size(int) \end{aligned} \tag{4}$$

Просуммировав все значения и приняв $size(\dots)$ за константы, можно сделать вывод, что $V(L_1, L_2) \in \Theta(L_1 \cdot L_2)$.

4.4.2 Оценка памяти реализации рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

Далее представлены типы данных и память, которую они занимают:

— Первая строка: $L_1 \cdot size(char)$;

— Вторая строка: $L_2 \cdot size(char)$;

— Размеры строк: $2 \cdot size(int)$;

— Указатели на строки и адрес возврата: $3 \cdot size(*)$;

— Вспомогательные переменные: $c \cdot size(int)$, где c — количество вспомогательных переменных;

Максимальная глубина стека вызовов при рекурсивной реализации алгоритма нахождения расстояния Дамерау — Левенштейна равна сумме длин входящих строк, соответственно, максимальный расход памяти рассчитывается по:

$$V(L_1, L_2) = (L_1 + L_2) \cdot ((2 + c) \cdot size(int) + 3 \cdot size(*)) \tag{5}$$

Просуммировав все значения и приняв $size(\dots)$ за константы, можно сделать вывод, что $V(L_1, L_2) \in \Theta(L_1 + L_2) = \Theta(L)$, то есть, максимальный расход памяти линейно зависит от длины входящих строк.

4.4.3 Оценка памяти реализации рекурсивного с кешированием алгоритма нахождения расстояния Дамерау — Левенштейна

Далее представлены типы данных и память, которую они занимают:

— Первая строка: $L_1 \cdot size(char)$;

— Вторая строка: $L_2 \cdot size(char)$;

— Размеры строк: $2 \cdot size(int)$;

— Указатели на строки и адрес возврата: $3 \cdot size(*)$;

— Матрица: $L_1 \cdot L_2 \cdot size(int) + L_1 \cdot size(*) + size(*)$;

— Вспомогательные переменные: $c \cdot size(int)$, где c — количество вспомогательных переменных;

Итого, под матрицу используется:

$$V_M(L_1, L_2) = L_1 \cdot L_2 \cdot size(int) + L_1 \cdot size(*) + size(*) \in \Theta(L_1 \cdot L_2) \qquad (6)$$

Под стек будет использоваться меньше памяти, чем в исключительно рекурсивной реализации, так как функция не будет углублять стек вызовов, если встретится уже вычисленное ранее значение. Следовательно, $V_S(L_1, L_2) \in \Theta(L)$.

Тогда суммарный объём используемой памяти имеет вид:

$$V(L_1, L_2) = V_M(L_1, L_2) + V_S(L_1, L_2) \in \Theta(L_1 \cdot L_2) + \Theta(L) = \Theta(L_1 \cdot L_2) \quad (7)$$

Таким образом, рекурсивная без кеширования реализация алгоритма нахождения Дамерау — Левенштейна расходует меньше памяти при достаточно длинных строках по сравнению с остальными реализациями. Её расход памяти растёт пропорционально $L_1 + L_2$, в то время как расход памяти остальных реализаций алгоритмов растёт пропорционально $L_1 \cdot L_2$.

4.5 Характеристики по памяти

На рисунках 4.9 – 4.12 представлены результаты измерения затрачиваемой памяти реализациями алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна. Замеры памяти проводились для строк одинаковой длины. Отображённая на графиках память является усреднённой для каждой длины строк.

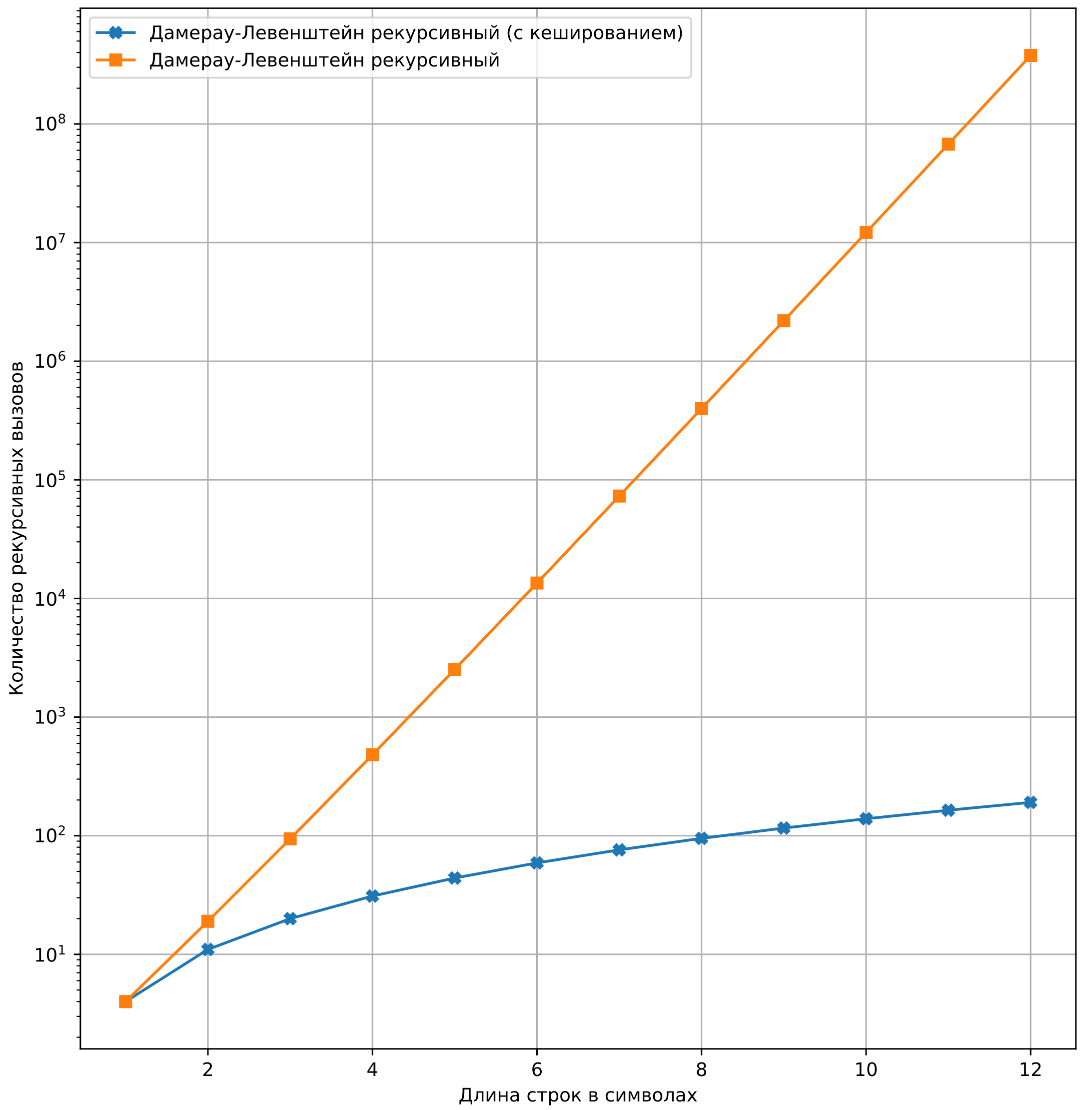

Рисунок 4.9 – Сравнение количества рекурсивных вызовов у рекурсивных реализаций алгоритмов нахождения расстояния Дамерау — Левенштейна

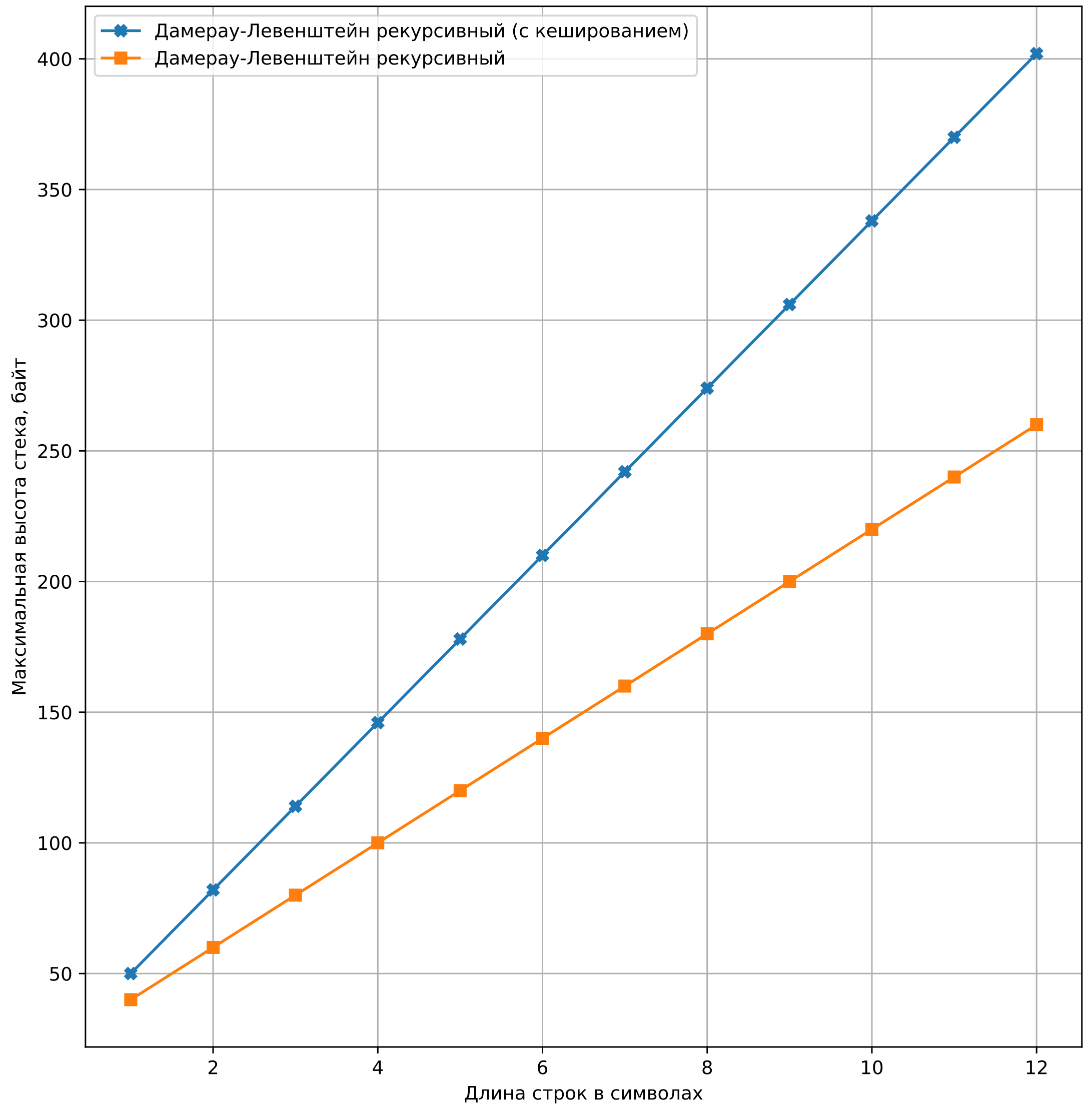

Рисунок 4.10 – Сравнение максимальной высоты стека вызовов у рекурсивных реализаций алгоритмов нахождения расстояния Дамерау — Левенштейна

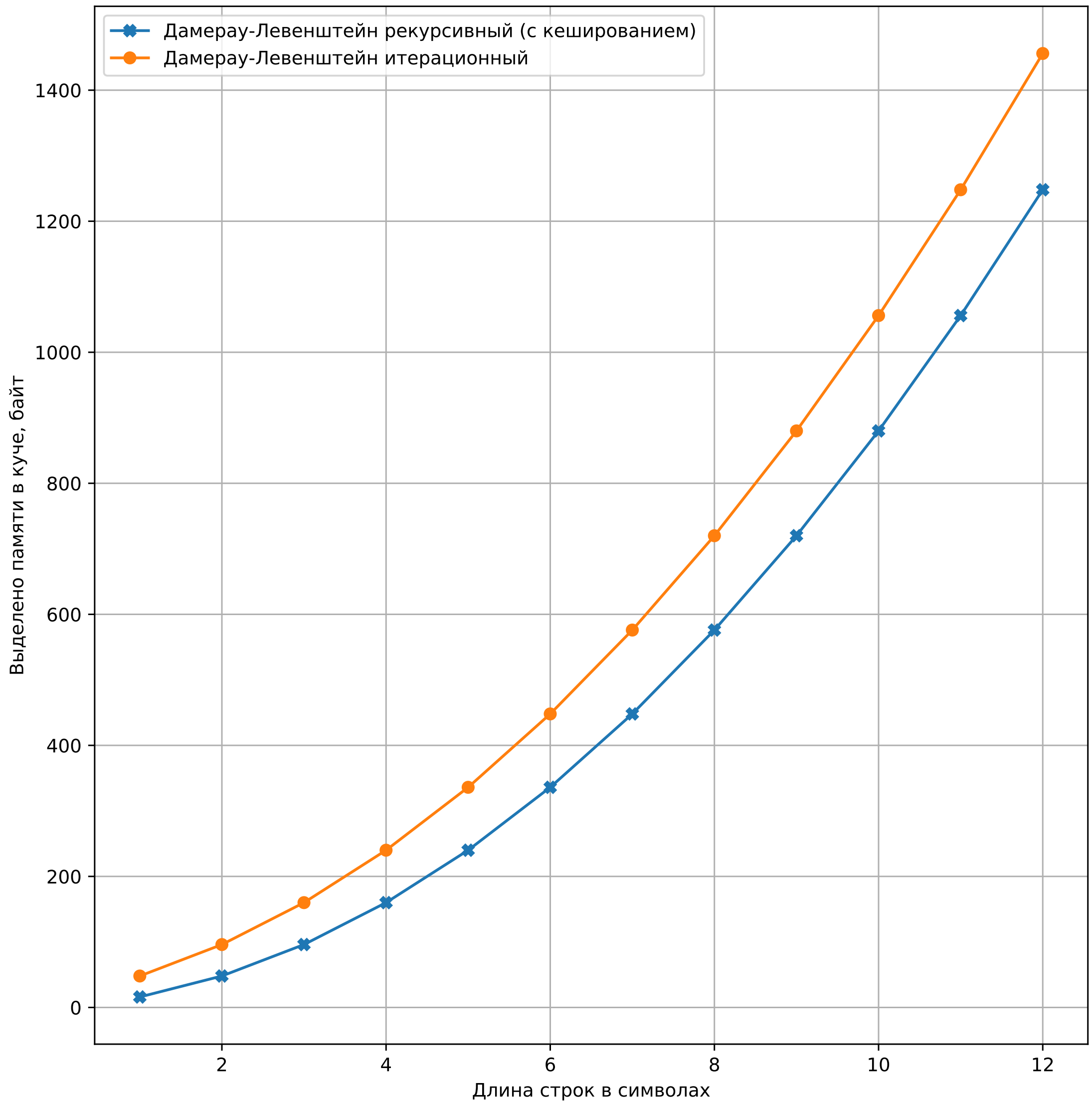

Рисунок 4.11 – Сравнение выделяемой памяти в куче для итерационного и рекурсивного с кешированием реализаций алгоритмов нахождения расстояния Дамерау — Левенштейна

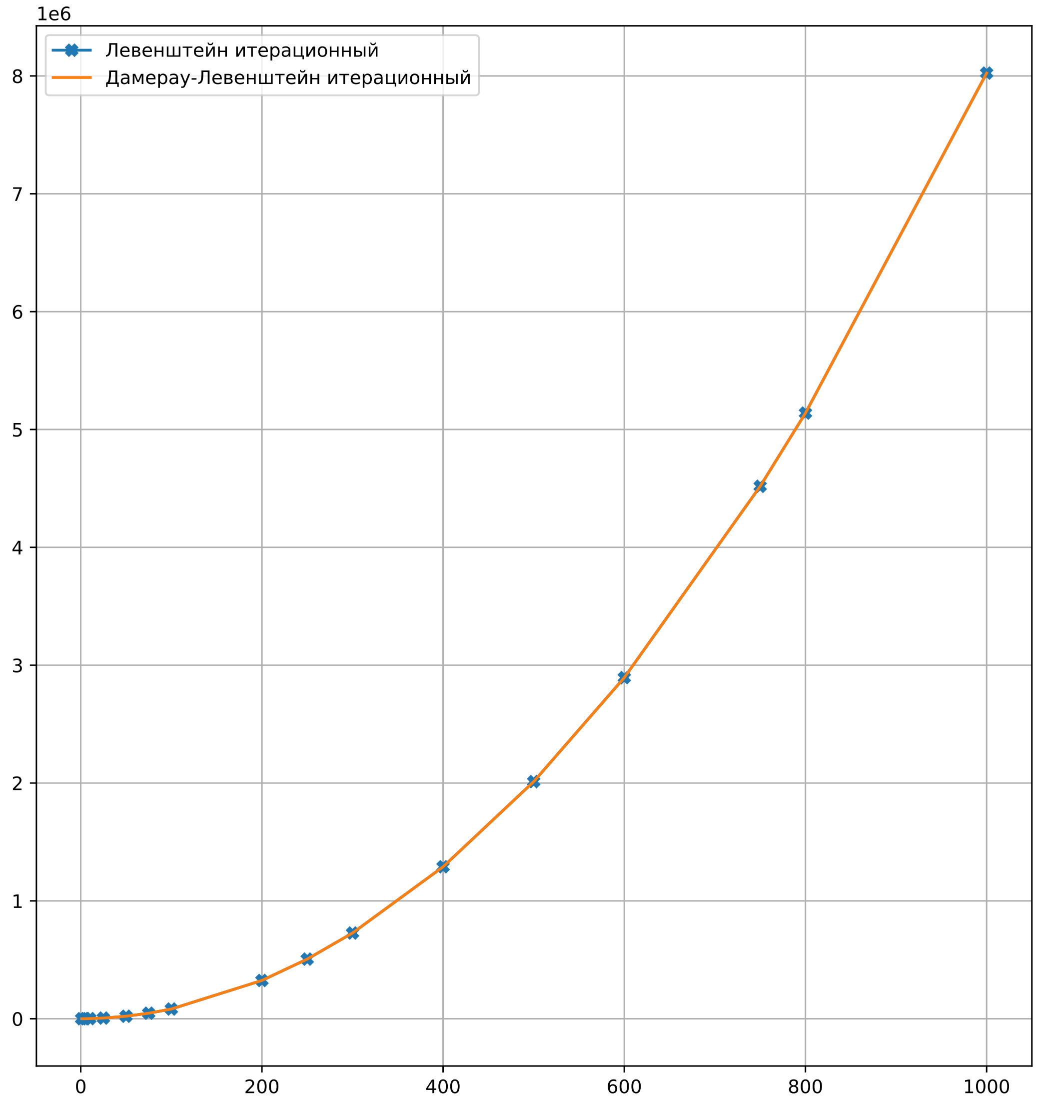

Рисунок 4.12 – Сравнение выделяемой памяти в куче для итерационных реализаций алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна

Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти реализаций алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна. Наименее затратной по времени оказалась итерационная реализация алгоритма нахождения расстояния Левенштейна.

Проанализировав использование памяти в алгоритмах, можно сделать вывод, что итеративные алгоритмы и рекурсивные алгоритмы с кешированием требуют больше памяти по сравнению с рекурсивным алгоритмом без кеширования. В реализациях, использующих матрицы, максимальный используемый объем памяти увеличивается пропорционально произведению длин строк. С другой стороны, для рекурсивного алгоритма без кеширования потребление памяти увеличивается пропорционально сумме длин строк.

ЗАКЛЮЧЕНИЕ

Цель данной работы была достигнута, а именно, были изучены, реализованы и исследованы алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна.

Для достижения поставленной цели были решены следующие задачи:

1) описаны алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна;
2) обоснован выбор средств реализации алгоритмов;
3) реализован итерационный алгоритм нахождения расстояния Левенштейна,
4) реализован итерационный алгоритм нахождения расстояния Дамерау — Левенштейна,
5) реализован рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна,
6) реализован рекурсивный с кешированием алгоритм нахождения расстояния Дамерау — Левенштейна;
7) проведён сравнительный анализ алгоритмов по критериям используемого процессорного времени и максимальной затрачиваемой памяти;
8) описаны и проанализированы полученные результаты в отчёте.

В результате исследования выяснилось, что наиболее эффективной по времени выполнения является итерационная реализация алгоритма нахождения расстояния Левенштейна. Она превосходит итерационную реализацию алгоритма нахождения расстояния Дамерау — Левенштейна в 1.1 раз, рекурсивную с кешированием — в 1.5 раз, рекурсивную без кеширования — в 865442 раза на строках длиной 12 символов.

Наименее затратным по памяти оказалась рекурсивная без кеширования реализация алгоритма нахождения расстояния Дамерау — Левенштейна. Она расходует память пропорционально сумме длин входящих строк, в то время как остальные реализации расходуют память пропорцонально произведению длин входящих строк.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. А. Погорелов Д., М. Таразанов А. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау — Левенштейна // Синергия Наук. 2019. — URL: https://elibrary.ru/item.asp?id=36907767 (дата обращения: 27.10.2023).

2. Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. — М.: «Наука», Доклады АН СССР, 1965. Т. 163. С. 845–848.

3. ISO C++ [Электронный ресурс]. — URL: https://isocpp.org/std/the-standard (дата обращения: 01.11.2023).

4. Linux man page — clock_gettime(3) [Электронный ресурс]. — URL: https://linux.die.net/man/3/clock_gettime (дата обращения: 01.11.2023).

5. GoogleTest User's Guide [Электронный ресурс]. — URL: https://google.github.io/googletest (дата обращения: 01.11.2023).

6. GDB: The GNU Project Debugger [Электронный ресурс]. — URL: https://www.sourceware.org/gdb (дата обращения: 01.11.2023).

7. Python [Электронный ресурс]. — URL: https://www.python.org (дата обращения: 01.11.2023).

8. Pandas [Электронный ресурс]. — URL: https://pandas.pydata.org (дата обращения: 01.11.2023).

9. NumPy [Электронный ресурс]. — URL: https://numpy.org (дата обращения: 01.11.2023).

10. Matplotlib [Электронный ресурс]. — URL: https://matplotlib.org (дата обращения: 01.11.2023).

11. Neovim [Электронный ресурс]. — URL: https://neovim.io (дата обращения: 01.11.2023).

12. AMD Ryzen™ 7 4700U [Электронный ресурс]. — URL: https://www.amd.com/en/product/9096 (дата обращения: 02.11.2023).

13. NixOS [Электронный ресурс]. — URL: https://nixos.org (дата обращения: 02.11.2023).

ПРИЛОЖЕНИЕ А

Листинг 2 – Создание матрицы для последующего использования в реализациях алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна

```
size_t **create_matrix(size_t n_rows, size_t n_columns)
{
    if (n_rows < 1 || n_columns < 1)
        return NULL;

    size_t *mem = (size_t*)malloc(n_rows * n_columns *
       sizeof(size_t));

    if (mem == NULL)
        return NULL;

    size_t **matrix = (size_t**)malloc(n_rows * sizeof(size_t*));

    if (matrix == NULL) {
        free(mem);
        return NULL;
    }

    for (size_t i = 0; i < n_rows; i++) {
        matrix[i] = mem + i * n_columns;
        matrix[i][0] = i;
    }

    for (size_t j = 1; j < n_columns; j++)
        matrix[0][j] = j;

    return matrix;
}
```

Листинг 3 – Освобождение памяти из-под созданной матрицы

```
void free_matrix(size_t **matrix, size_t *first_row)
{
    free(first_row);
    free(matrix);
}
```

Листинг 4 – Нахождение минимального числа из трёх и возврат указателя на него

```
size_t *min3(size_t *a, size_t *b, size_t *c)
{
    size_t *result;
    size_t min = std::min(*a, std::min(*b, *c));

    if (min == *a)
        result = a;
    else if (min == *b)
        result = b;
    else
        result = c;

    return result;
}
```

Листинг 5 – Нахождение минимального числа из четырёх и возврат указателя на него

```
size_t *min4(size_t *a, size_t *b, size_t *c, size_t *d)
{
    size_t *result;
    size_t min = std::min(std::min(*a, *b), std::min(*c, *d));

    if (min == *a)
        result = a;
    else if (min == *b)
        result = b;
    else if (min == *c)
        result = c;
    else
        result = d;

    return result;
}
```

Листинг 6 – Часть реализации итерационного алгоритма нахождения расстояния Левенштейна, участвующая в замерах времени выполнения реализаций исследуемых алгоритмов

```
size_t lev_ifm_helper(size_t **matrix, const wchar_t *s1, size_t
    len1, const wchar_t *s2, size_t len2)
{
    size_t result = 0;
    bool replace_skip_cond;
    size_t insert_cost, delete_cost, replace_cost, *who;

    for (size_t i = 1; i < len1; i++) {
        for (size_t j = 1; j < len2; j++) {
            insert_cost = matrix[i - 1][j] + 1;
            delete_cost = matrix[i][j - 1] + 1;
            replace_skip_cond = (s1[i] == s2[j]);
            replace_cost = matrix[i - 1][j - 1] +
                (replace_skip_cond ? 0 : 1);
            who = min3(&insert_cost, &delete_cost, &replace_cost);
            matrix[i][j] = *who;
        }
    }

    result = matrix[len1 - 1][len2 - 1];

    return result;
}
```

Листинг 7 – Реализация итерационного алгоритма нахождения расстояния Левенштейна

```
size_t levenshtein_iterative_full_matrix(const wchar_t *str1,
    size_t len1, const wchar_t *str2, size_t len2)
{
    if (len1 == 0) return len2;
    if (len2 == 0) return len1;

    ++len1;
    ++len2;
    const wchar_t *s1 = str1 - 1;
    const wchar_t *s2 = str2 - 1;

    size_t **matrix = create_matrix(len1, len2);

    if (matrix == NULL) return -1;

    size_t result = lev_ifm_helper(matrix, s1, len1, s2, len2);

    free_matrix(matrix, matrix[0]);

    return result;
}
```

Листинг 8 – Часть реализации итерационного алгоритма нахождения расстояния Дамерау — Левенштейна, участвующая в замерах времени выполнения реализаций исследуемых алгоритмов

```
size_t damlev_ifm_helper(size_t **matrix, const wchar_t *s1, size_t len1, const wchar_t *s2, size_t len2)
{
    size_t result = 0;
    bool replace_skip_cond, swap_cond;
    size_t insert_cost, delete_cost, replace_cost, swap_cost, *who;

    for (size_t i = 1; i < len1; i++) {
        for (size_t j = 1; j < len2; j++) {
            insert_cost = matrix[i - 1][j] + 1;
            delete_cost = matrix[i][j - 1] + 1;
            replace_skip_cond = (s1[i] == s2[j]);
            replace_cost = matrix[i - 1][j - 1] + (replace_skip_cond ? 0 : 1);
            if (i >= 2 && j >= 2) [[likely]] {
                swap_cond = (s1[i] == s2[j - 1] && s1[i - 1] == s2[j]);
                swap_cost = swap_cond ? matrix[i - 2][j - 2] + 1 : U_INF; // U_INF = -1
                who = min4(&insert_cost, &delete_cost, &replace_cost, &swap_cost);
            } else {
                who = min3(&insert_cost, &delete_cost, &replace_cost);
            }
            matrix[i][j] = *who;
        }
    }

    result = matrix[len1 - 1][len2 - 1];

    return result;
}
```

Листинг 9 – Реализация итерационного алгоритма нахождения расстояния Дамерау — Левенштейна

```
size_t damerau_levenshtein_iterative_full_matrix(const wchar_t
    *str1, size_t len1, const wchar_t *str2, size_t len2)
{
    if (len1 == 0) return len2;
    if (len2 == 0) return len1;

    ++len1;
    ++len2;
    const wchar_t *s1 = str1 - 1;
    const wchar_t *s2 = str2 - 1;

    size_t **matrix = create_matrix(len1, len2);
    if (matrix == NULL) return -1;

    size_t result = damlev_ifm_helper(matrix, s1, len1, s2, len2);

    free_matrix(matrix, matrix[0]);

    return result;
}
```

Листинг 10 – Часть реализации рекурсивного с кешированием алгоритма нахождения расстояния Дамерау — Левенштейна, участвующая в замерах времени выполнения реализаций исследуемых алгоритмов

```
size_t damlev_rwc_helper(size_t **matrix, const wchar_t *str1,
    size_t len1, const wchar_t *str2, size_t len2)
{
    if (len1 == 0) return len2;
    if (len2 == 0) return len1;

    size_t i = len1 - 1;
    size_t j = len2 - 1;

    size_t insert = (((j > 0) && (matrix[i][j - 1] != U_INF))
        ? matrix[i][j - 1]
        : damlev_rwc_helper(matrix, str1, len1, str2, len2 - 1))
        + 1;

    size_t del = (((i > 0) && (matrix[i - 1][j] != U_INF))
        ? matrix[i - 1][j]
        : damlev_rwc_helper(matrix, str1, len1 - 1, str2, len2))
        + 1;

    size_t replace = (((i > 0 && j > 0) && (matrix[i - 1][j - 1]
        != U_INF))
        ? matrix[i - 1][j - 1]
        : damlev_rwc_helper(matrix, str1, len1 - 1, str2, len2 -
            1))
        + (str1[i] == str2[j] ? 0 : 1);

    size_t swap = U_INF;
    if (i > 1 && j > 1 && matrix[i - 2][j - 2] != U_INF &&
        (str1[i] == str2[j - 1] && str1[i - 1] == str2[j])) {
        swap = matrix[i - 2][j - 2] + 1;
    } else if (i >= 1 && j >= 1 && (str1[i] == str2[j - 1] &&
        str1[i - 1] == str2[j])) {
        swap = damlev_rwc_helper(matrix, str1, len1 - 2, str2,
            len2 - 2) + 1;
    }

    size_t result = *min4(&insert, &del, &replace, &swap);

```

```
        if (matrix[i][j] == U_INF) matrix[i][j] = result;

        return result;
}
```

Листинг 11 – Реализация рекурсивного с кешированием алгоритма нахождения расстояния Дамерау — Левенштейна

```
size_t damerau_levenshtein_recursive_with_cache(const wchar_t
    *str1, size_t len1, const wchar_t *str2, size_t len2)
{
    if (len1 == 0) return len2;
    if (len2 == 0) return len1;

    size_t **matrix = create_matrix(len1, len2);

    if (matrix == NULL) return -1;

    for (size_t i = 0; i < len1; i++)
        for (size_t j = 0; j < len2; j++)
            matrix[i][j] = U_INF;

    size_t result = damlev_rwc_helper(matrix, str1, len1, str2,
        len2);

    free_matrix(matrix, matrix[0]);

    return result;
}
```

Листинг 12 – Реализация рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

```
size_t damerau_levenshtein_recursive_no_cache(const wchar_t *str1, size_t len1, const wchar_t *str2, size_t len2)
{
    if (len1 == 0) return len2;
    if (len2 == 0) return len1;

    size_t insert = damerau_levenshtein_recursive_no_cache(str1, len1, str2, len2 - 1) + 1;
    size_t del = damerau_levenshtein_recursive_no_cache(str1, len1 - 1, str2, len2) + 1;
    size_t replace = damerau_levenshtein_recursive_no_cache(str1, len1 - 1, str2, len2 - 1)
        + (str1[len1 - 1] == str2[len2 - 1] ? 0 : 1);
    size_t swap = (len1 >= 2 && len2 >= 2)
        ? (
                (str1[len1 - 1] == str2[len2 - 2] && str1[len1 - 2] == str2[len2 - 1])
                ? damerau_levenshtein_recursive_no_cache(str1, len1 - 2, str2, len2 - 2) + 1
                : U_INF
          )
        : U_INF;

    return *min4(&insert, &del, &replace, &swap);
}
```