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"

Тема Расстояния Левенштейна и Дамерау-Левенштейна

Студент Хамзина Р. Р.

Группа ИУ7-53Б

Оценка (баллы)

Преподаватель Волкова Л. Л.

Москва — 2021 г.

Содержание

Введение

Задача поиска и сравнения последовательностей возникает при обработке больших объемов информации, при работе с неструктурированными данными или при поисковых запросах. Алгоритмы для решения этой задачи разделяются на три группы: точный поиск подстрок, неточный поиск и поиск наибольшей общей подпоследовательности. Алгоритмы нечеткого поиска применяются в компьютерной лингвистике (коррекция ошибок, поиск с учетом формы одного и того же слова) и в биоинформатике (сравнение ДНК). В основе методов лежит определение расстояния между строками, которое задается в конкретном алгоритме.

Данная задача впервые была поставлена советским математиком Владимиром Левенштейном для двоичных кодов. Расстояние между строками для прозвольного алфавита назвали расстоянием Левенштейна и определили как минимальное количество операций вставки, удаления и замены одного символа, необходимых для превращения одной строки в другую [1].

Модификацией алгоритма Левенштейна является алгоритм Дамерау-Левенштейна. В данном алгоритме к операциям, необходимым для перевода одной строки в другую, добавляется транспозиция символов (перестановка двух соседних символов).

Данные алгоритмы реализуются методом динамического программирования. Существуют их рекурсивная и нерекурсивная реализации, которые различаются временной эффективностью.

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Для достижения поставленной цели требуется выполнить следующие задачи:

- изучить алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- привести схемы изучаемых алгоритмов;
- описать используемые типы и структуры данных;
- оценить объем памяти для хранения данных;

- описать структуру разрабатываемого программного обеспечения;
- определить средства программной реализации выбранных алгоритмов;
- реализовать разработанные алгоритмы;
- провести функциональное тестирование программного обеспечения;
- провести сравнительный анализ по времени реализованных алгоритмов;
- подготовить отчет о выполненной лабораторной работе.

1 Аналитическая часть

В данном разделе будут описаны алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Рассмотрим перевод строки $A' = A[:i]$ в строку $B' = B[:j]$, последние символы которых соответственно a_{i-1} и b_{j-1}, при помощи следующих операций, каждая из которых имеет свою стоимость (x, y - символы, λ - строка):

- вставка символа в произвольное место - I, стоимостью $w(x, \lambda) = 1$;
- удаление символа с произвольной позиции - D, $w(\lambda, y) = 1$;
- замена символа на другой - R, $w(x, y) = 1$, $x \neq y$.

Расстояние Левенштейна - минимальное количество операций I, D, R для перевода $A' = A[:i]$ в $B' = B[:j]$ [1]. Его можно подсчитать по следующей реккурентной формуле:

$$D(i,j) = \begin{cases} 0 & \text{i} = 0, \text{j} = 0 \\ i & \text{j} = 0, \text{i} > 0 \\ j & \text{i} = 0, \text{j} > 0 \\ \min\{ & \\ \quad D(i, j-1) + 1 & \\ \quad D(i-1, j) + 1 & \text{i} > 0, \text{j} > 0 \\ \quad D(i-1, j-1) + m(a_{i-1}, b_{j-1}) & (2.1) \\ \} & \end{cases} \tag{1.1}$$

Функция 2.1 определена как:

$$m(x,y) = \begin{cases} 0 & \text{если x = y,} \\ 1 & \text{иначе} \end{cases}. \tag{1.2}$$

Рекурсия используется для реализации формулы 1.1. Пусть $D(i,j)$ - есть расстояние Левенштейна между строками $A' = A[:i]$ и $B' = B[:j]$. $B' = B[:j]$ может быть получен из $A' = A[:i]$ следующими способами:

- если символ a_{j-1} был добавлен при редактировании, то необходимо $A[:i]$ превратить в $B[:j-1]$, на что необходимо $D(i,j-1)$ операций редактирования. В этом случае $D(i,j) = D(i,j-1) + 1$;
- если символ a_{i-1} был удален при редактировании, тогда необходимо $A[:i-1]$ превратить в $B[:j]$, на что необходимо $D(i-1,j)$ операций редактирования. В этом случае $D(i,j) = D(i-1,j) + 1$;
- если последние символы префиксов совпадают, т. е. $a_{i-1} = b_{j-1}$, то в этом случае можно не менять эти последние символы. Тогда $D(i,j) = D(i-1,j-1)$;
- если $a_{i-1} \neq b_{j-1}$, то тогда можно потратить 1 операцию на замену символа a_{i-1} на b_{j-1} и также потратить операцию $D(i-1,j-1)$ на превращение $A[:i-1]$ в $B[:j-1]$. Тогда $D(i,j) = D(i-1,j-1) + 1$.

Далее при вычислении необходимо взять минимум из всех перечисленных возможностей (при этом из случаев 3 или 4 рассматривается только один, в зависимости от условия $a_{i-1} = b_{j-1}$).

Начальные значения: $D(i,0) = i$, $D(0,j) = j$.

1.2 Матричный алгоритм нахождения расстояния Левенштейна

Промежуточные значения $D(i,j)$ вычисляются несколько раз, поэтому при больших значениях i, j временная эффективность алгоритма сни-

жается. Для решения этой проблемы используют матрицу для хранения значений расстояний.

Матрицу M размером $N+1$ и $L+1$, где N и L - длины строк $A' = A[:i]$ и $B' = B[:j]$, заполняют по следующим правилам:

- $M[0][0] = 0$;
- $M[i][j]$ вычисляется по формуле 1.3.

$$M[i][j] = \min \begin{cases} M[i][j-1] + 1 \\ M[i-1][j] + 1 \\ M[i-1][j-1] + m(A[:i-1], B[:j-1]) \quad (2.2) \end{cases} \tag{1.3}$$

Функция 2.2 определена как:

$$m(x, y) = \begin{cases} 0 & \text{если x = y,} \\ 1 & \text{иначе} \end{cases}. \tag{1.4}$$

$M[N][L]$ содержит значение расстояния Левенштейна.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием матрицы

Для оптимизации рекурсивного алгоритма можно использовать матрицу как промежуточный буфер, в который при выполнении рекурсии заносятся значения расстояния Левенштейна для еще не рассмотренных символов. Рассмотренные раннее символы в матрицу не добавляются.

1.4 Алгоритм нахождения расстояния Дамерау-Левенштейна

Как было сказано раннее, алгоритм Дамерау-Левенштейна отличается от алгоритма Левенштейна добавлением операции транспозиции символов при переводе одной строки в другую. Расстояние Дамерау-Левенштейна может быть найдено по формуле:

$$d(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d(i,j-1)+1, & \\ \quad d(i-1,j)+1, & \\ \quad d(i-1,j-1)+m(a_{i-1},b_{j-1}), & \text{иначе} \\ \quad \begin{cases} d(i-2,j-2)+1, & \text{если } i,j>1; \\ & a_i = b_{j-1}; \\ & b_j = a_{i-1} \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases}, \tag{1.5}$$

Пусть $d(i,j)$ - есть расстояние Дамерау-Левенштейна между строками $A' = A[:i]$ и $B' = B[:j]$. $B' = B[:j]$ может быть получен из $A' = A[:i]$ следующими способами:

- если символ a_{i-1} был удален при редактировании, тогда необходимо $A[:i-1]$ превратить в $B[:j]$, на что необходимо $d(i-1,j)$ операций редактирования. В этом случае $d(i,j) = d(i-1,j)+1$;
- если символ a_{j-1} был добавлен при редактировании, то необходимо $A[:i]$ превратить в $B[:j-1]$, на что необходимо $d(i,j-1)$ операций редактирования. В этом случае $d(i,j) = d(i,j-1)+1$;
- если последние символы префиксов совпадают, т. е. $a_{i-1} = b_{j-1}$, то в

этом случае можно не менять эти последние символы. Тогда $d(i,j) = d(i-1,j-1)$;

- если $a_{i-1} \neq b_{j-1}$, то тогда можно потратить 1 операцию на замену символа a_{i-1} на b_{j-1} и также потратить операцию $D(i-1,j-1)$ на превращение $A[:i-1]$ в $B[:j-1]$. Тогда $d(i,j) = d(i-1,j-1)+1$;

- если последние два символа a_i и b_j были переставлены, то $d(i,j)$ может быть равно $d(i-2,j-2)+1$.

Далее при вычислении необходимо взять минимум из всех перечисленных возможностей (при этом из случаев 3 или 4 рассматривается только один, в зависимости от условия $a_{i-1} = b_{j-1}$).

Начальные значения: $D(i,0) = i$, $D(0,j) = j$.

1.5 Вывод

Были изучены способы нахождения расстояния перевода одной строки в другую при помощи рекурсивной и матричной версий алгоритма Левенштейна, алгоритма Левенштейна с использованием матрицы и рекурсивной версии алгоритма Дамерау-Левенштейна.

Программе, реализующей данные алгоритмы, на вход будет подаваться две строки. Выходными данными такой программы должно быть число - редакционное расстояние. Программа должна работать в рамках следующих ограничений: строки, поданные на вход, могут состоять из букв русского и английского языков. Кроме того, должен быть выдан корректный результат при вводе пустых строк.

Пользователь должен иметь возможность выбора алгоритма нахождения редакционного расстояния и вывода его значения на экран. Также должны быть реализованы сравнение алгоритмов по времени работы с выводом результатов на экран и получение графического представления результатов сравнения. Данные действия пользователь должен выполнять при помощи меню.

2 Конструкторская часть

В данном разделе будут представлены схемы алгоритмов вычисления расстояний Левенштейна и Дамерау-Левенштейна. Будут описаны типы и структуры данных, используемые для реализации, а также структура разрабатываемого программного обеспечения, оценено использование памяти. Кроме того будут выделены классы эквивалентности для тестирования.

2.1 Разработка алгоритмов

Схемы рекурсивного и матричного алгоритмов поиска расстояния Левенштейна представлены на рисунках **??**-**??** соответственно. Для алгоритма с использованием матрицы на рисунке **??** приведена схема рекурсивной части, схема всего алгоритма - на рисунке **??**. Схема алгоритма Дамерау-Левенштейна приведена на рисунке **??**.

2.2 Описание используемых типов и структур данных

Для реализации рассмотренных алгоритмов будут использованы следующие типы данных:

- *str* - для двух строк, поданных на вход;
- *int* - для длин каждой из строк, поданных на вход, и для возвращаемого значения искомого расстояния.

Для реализации матричной версии алгоритма нахождения расстояния Левенштейна и версии с использованием матрицы используется двумерный список значений типа *int*.

2.3 Использование памяти

Алгоритм нахождения расстояния Дамерау-Левенштейна отличается от алгоритма нахождения расстояния Левенштейна добавлением операции транспозиции, что не влияет на объем используемой памяти. Поэтому проведем анализ используемой памяти для рекурсивного и нерекурсивного версий алгоритма Левенштейна.

Для рекурсивной версии при каждом вызове происходит выделение памяти под:

- 2 строки типа str;
- 2 длины строк типа int;
- возвращаемое значение типа int;
- адрес возврата типа int;
- локальная переменная типа int.

При этом глубина рекурсии равна сумме длин двух строк. Таким образом, используемая память рекурсивной версии равна:

$$M_r = (5 \cdot sizeof(int) + 2 \cdot sizeof(str)) \cdot (|s1| + |s2|) \tag{2.1}$$

Для нерекурсивной версии происходит выделение памяти под:

- 2 строки типа str;
- 2 длины строк типа int;
- матрицу с размерами, равными длинам строк, увеличенным на 1;
- возвращаемое значение типа int;
- адрес возврата типа int;
- 5 локальных переменных типа int.

Таким образом, используемая память нерекурсивной версии равна:

$$M_m = (9 + (|s1| + 1) \cdot (|s2| + 1)) \cdot sizeof(int) + 2 \cdot sizeof(str) \qquad (2.2)$$

2.4 Структура разрабатываемого ПО

Программа состоит из следующих модулей:

- main.py - главный файл программы, предоставляющий пользователю меня для выполнения основных функций;
- distances.py - файл, содержащий алгоритмы нахождения редакционного расстояния;
- str.py - файл, содержащий функции работы со строками;
- matrix.py - файл, содержащий функции работы с матрицами;
- time_test.py - файл, содержащий функции замеров времени работы указанных алгоритмов;
- graph_result.py - файл, содержащий функции визуализации временных характеристик описанных алгоритмов.

2.5 Классы эквивалентности при тестировании

Для тестирования разрабатываемой программы будут выделены следующие классы эквивалентности:

- две пустые строки;
- одна пустая строка, одна нет;
- удаление/добавление символа в конец;

- удаление/добавление символа внутри строки;
- замена символов;
- транспозиция символов.

2.6 Вывод

Были представлены схемы алгоритмов вычисления расстояний Левенштейна и Дамерау-Левенштейна. Были указаны типы и структуры данных, используемые для реализации, и описана структура разрабатываемого программного обеспечения. Был проанализирован объем используемой памятия для рекурсивной и нерекурсивной версий алгоритмов. Также были выделены классы эквивалентности для тестирования ПО.

3 Технологическая часть

В данном разделе будут указаны средства реализации, будут представлены листинги кода, а также функциональные тесты.

3.1 Средства реализации

Реализация данной лабораторной работы выполнялась при помощи языка программирования Python [2]. Выбор ЯП обусловлен простотой синтаксиса, большим числом библиотек и эффективностью визуализации данных.

Замеры времени проводились при помощи функции process_time из библиотеки time [3].

3.2 Листинги кода

Стандартный алгоритм, алгоритм Винограда и оптимизированный алгоритм Винограда умножения матриц приведены в листингах 3.1-3.3.

Листинг 3.1 – Стандартный алгоритм умножения матриц

```
def standard_mult(A, B):
    n = len(A)
    m = len(B[0])
    p = len(A[0])

    C = [[0] * m for i in range(n)]

    for i in range(n):
        for j in range(m):
            for k in range(p):
                C[i][j] += A[i][k] * B[k][j]

    return C
```

Листинг 3.2 – Алгоритм Винограда умножения матриц

```
def winograd_mult(A, B):
```

```
    n = len(A)
    m = len(B[0])
    p = len(A[0])

    C = [[0] * m for _ in range(n)]

    row_factors = [0] * n

    for i in range(n):
        for j in range(p // 2):
            row_factors[i] = row_factors[i] + \
                             A[i][2 * j] * A[i][2 * j + 1]

    column_factors = [0] * m

    for i in range(m):
        for j in range(p // 2):
            column_factors[i] = column_factors[i] + \
                                B[2 * j][i] * B[2 * j + 1][i]

    for i in range(n):
        for j in range(m):
            C[i][j] = -row_factors[i] - column_factors[j]

            for k in range(p // 2):
                C[i][j] = C[i][j] + (A[i][2 * k + 1] + B[2 *
                   k][j]) * \
                                    (A[i][2 * k] + B[2 * k +
                                       1][j])

    if p % 2 != 0:
        for i in range(n):
            for j in range(m):
                C[i][j] = C[i][j] + A[i][p - 1] * B[p - 1][j]

    return C
```

Листинг 3.3 – Оптимизированный алгоритм Винограда умножения матриц

```
def optimized_winograd_mult(A, B):
    n = len(A)
```

```
3       m = len(B[0])
4       p = len(A[0])
5
6       C = [[0] * m for _ in range(n)]
7
8       row_factors = [0] * n
9
10      for i in range(n):
11          for j in range(1, p, 2):
12              row_factors[i] += A[i][j] * A[i][j - 1]
13
14      column_factors = [0] * m
15
16      for i in range(m):
17          for j in range(1, p, 2):
18              column_factors[i] += B[j][i] * B[j - 1][i]
19
20      flag = p % 2
21
22      for i in range(n):
23          for j in range(m):
24              C[i][j] = -(row_factors[i] + column_factors[j])
25
26              for k in range(1, p, 2):
27                  C[i][j] += (A[i][k - 1] + B[k][j]) * \
28                                          (A[i][k] + B[k - 1][j])
29
30              if flag:
31                  C[i][j] += A[i][p - 1] * B[p - 1][j]
32
33      return C
```

3.3 Функциональные тесты

В таблице ?? приведены функциональные тесты для функций, реализующих алгоритмы умножения матриц. Все тесты пройдены успешно.

3.4 Вывод

Были реализованы функции алгоритмов умножения матриц. Было проведено функциональное тестирование указанных функций.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программы, и будет проведен сравнительный анализ реализованных алгоритмов умножения матриц по затраченному процессорному времени.

4.1 Технические характеристики

Тестирование проводилось на устройстве со следующими техническими характеристиками:

- операционная система: Ubuntu 20.04.1 Linux x86_64 [4];
- память : 8 GiB;
- процессор: AMD® Ryzen™ 3 3200u @ 2.6 GHz [5].

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 приведен пример работы программы.

```
МЕНЮ:
  1 - стандартное умножение матриц;
  2 - умножение матриц по алгоритму Винограда;
  3 - умножение матриц по оптимизированному алгоритму Винограда;
  4 - построить графики;
  5 - замерить время;
  0 - выход.
Выбор: 3

Введите матрицу A!
Введите число строк: 3
Введите число столбцов: 2
1
2
3
4
5
6

Введенная матрица A:
 1 2
 3 4
 5 6

Введите матрицу B!
Введите число строк: 2
Введите число столбцов: 4
1
2
3
4
5
6
7
8

Введенная матрица B:
 1 2 3 4
 5 6 7 8

Полученная матрица:
 11 14 17 20
 23 30 37 44
 35 46 57 68
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Функция process_time из библиотеки time ЯП Python возвращает процессорное время в секундах - значение типа float.

Для замера времени:

- получить значение времени до начала выполнения алгоритма, затем после её окончания. Чтобы получить результат, необходимо вычесть из второго значения первое;
- первый шаг необходимо повторить iters раз (в программе iters равно 10), суммируя полученные значения, а затем усреднить результат.

Замеры проводились для квадратных матриц целых чисел, заполненных случайным образом, размером от 10 до 100 и от 11 до 110. Результаты измерения времени для четного размера матриц приведены в таблице 4.1 (в мс).

Таблица 4.1 – Результаты замеров времени

Размер	Стандартный	Винограда	Опт-ый Винограда
10	0.2875	0.6851	0.2061
20	1.6220	3.5965	1.2381
30	5.3519	4.4189	4.0736
40	12.2201	9.7817	9.0212
50	23.4212	18.8085	18.8640
60	39.3355	32.1766	30.2312
70	61.6586	52.5354	50.6840
80	91.7189	81.2931	73.1704
90	129.2349	110.4791	104.2271
100	177.5043	158.5571	134.2171

На рисунке 4.2 приведены графические результаты сравнения временных характеристик для четного размера матриц.

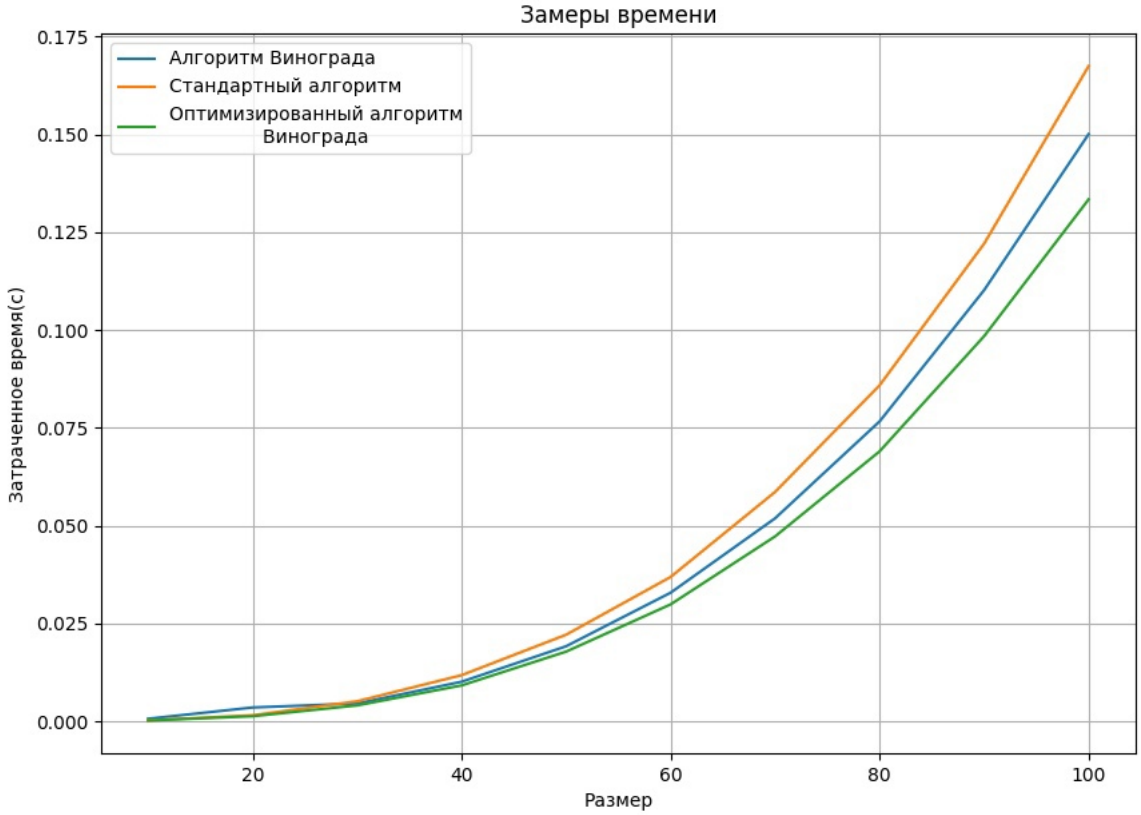

Рисунок 4.2 – Сравнение по времени алгоритмов умножения матриц четного размера

Результаты измерения времени для четного размера матриц приведены в таблице 4.2 (в мс).

Таблица 4.2 – Результаты замеров времени

Размер	Стандартный	Винограда	Опт-ый Винограда
11	0.2942	0.8884	0.2907
21	1.8007	3.2802	1.6107
31	5.5211	4.6575	4.8589
41	12.0378	10.6847	10.6740
51	23.3503	20.2865	19.9010
61	41.0413	34.2695	32.9210
71	65.2536	54.6966	50.5432
81	93.8148	82.2679	72.8342
91	131.0258	118.6902	105.8783
101	177.7476	158.8673	142.5004

На рисунке 4.3 приведены графические результаты сравнения временных характеристик для нечетного размера матриц.

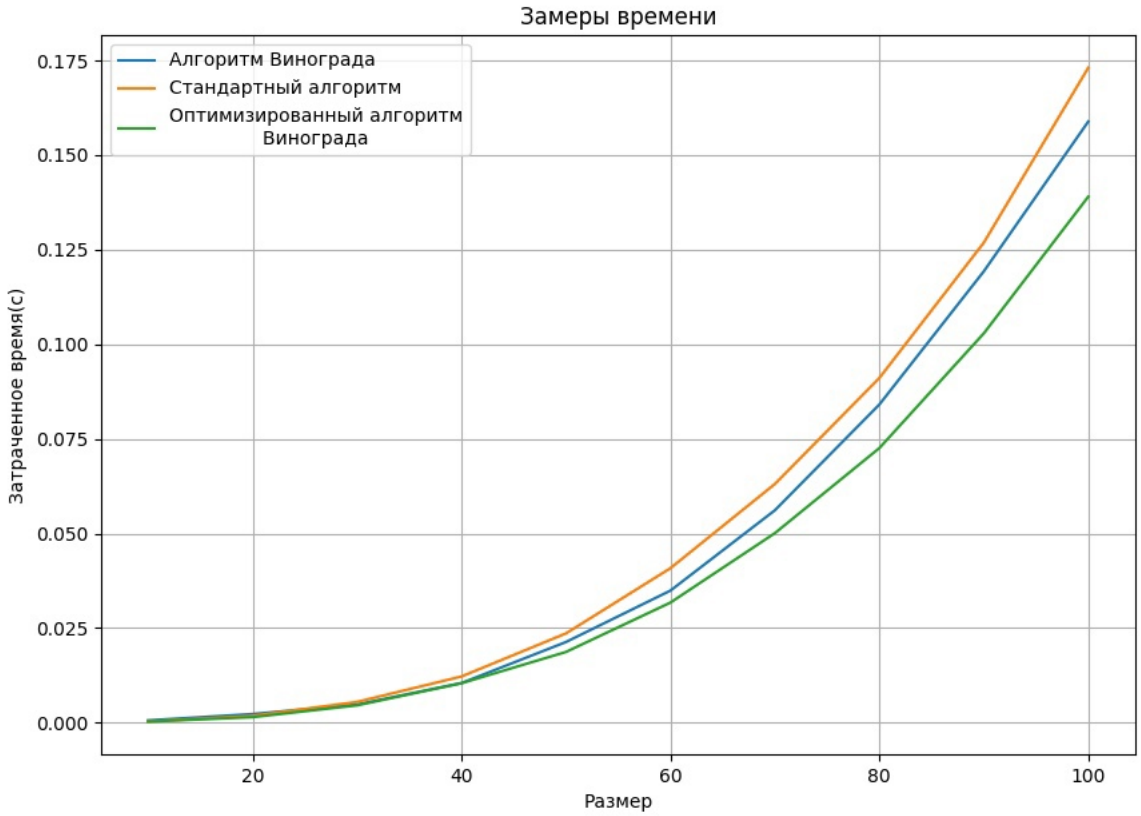

Рисунок 4.3 – Сравнение по времени матричного алгоритма Левенштейна и алгоритма Левенштейна с использованием кэша

4.4 Вывод

Исходя из замеров по памяти, итеративные алгоритмы проигрывают рекурсивным, потому что максимальный размер памяти в них растет, как произведение длин строк, а в рекурсивных - как сумма длин строк.

В результате эксперимента было получено, что при длине строк в 4 символа, рекурсивная реализация алгоритма Левенштейна становится медленнее матричной в 9 раз и с увеличением длины строк время работы растет в геометрической прогрессии. Тогда, для строк длиной более 4 символов небходимо использовать матричную версию алгоритма поиска редакционного расстояния.

Также в результате эксперимента было установлено, что при длине строк в более 5 символов, алгоритм Левенштейна работает быстрее Дамерау-Левенштейна в 1.3 раза. Можно сделать вывод, что при таких данных пред-

почтительно использовать алгоритм Левенштейна.

Заключение

В результате оценки алгоритмов по используемой памяти можно сделать вывод, что матричные реализации алгоритмов поиска редакционного расстояния занимают больше памяти, чем рекурсивные, так как память, используемая нерекурсивной реализацией растет как произведение длин строк, а память, используемая рекурсивной реализацией растет как сумма длин строк.

В связи с дополнительной операцией алгоритм Дамерау-Левенштейна работает в 1.3 раза медленнее, алгоритма Левенштейна. При этом среди реализаций алгоритмов подсчет расстояний Левенштейна быстрые результаты дает рекурсивная реализация с использованием матрицы за счет сохранения в ней промежуточных значений.

Цель, поставленная перед началом работы, была достигнута. В ходе лабораторной работы были решены следующие задачи:

- были изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- были разработаны изученные алгоритмы;
- был проведен сравнительный анализ реализованных алгоритмов;
- был подготовлен отчет о выполненной лабораторной работе.

Список литературы

[1] Вычисление расстояния Левенштейна [Электронный ресурс]. Режим доступа: https://foxford.ru/wiki/informatika/vychislenie-rasstoyaniya-levenshteyna (дата обращения: 18.10.2021).

[2] Welcome to Python [Электронный ресурс]. Режим доступа: https://www.python.org (дата обращения: 18.10.2021).

[3] time — Time access and conversions [Электронный ресурс]. Режим доступа: https://docs.python.org/3/library/time.html#functions (дата обращения: 18.10.2021).

[4] Ubuntu 20.04 LTS (Focal Fossa) Beta [Электронный ресурс]. Режим доступа: http://old-releases.ubuntu.com/releases/20.04.1/ (дата обращения: 18.10.2021).

[5] Мобильный процессор AMD Ryzen™ 3 3200U с графикой Radeon™ Vega 3 [Электронный ресурс]. Режим доступа: https://www.amd.com/ru/products/apu/amd-ryzen-3-3200u (дата обращения: 18.10.2021).