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 № 2

по курсу «Анализ Алгоритмов»

на тему: «Умножение матриц (сложность)»

Студент ИУ7-53Б _____ Лысцев Н. Д.
(Группа) (Подпись, дата) (И. О. Фамилия)

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

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ **3**

1 Аналитический раздел **4**

1.1 Матрица 4

1.2 Классический алгоритм умножения двух матриц 5

1.3 Алгоритм Винограда для умножения двух матриц 5

1.4 Оптимизированный алгоритм Винограда для умножения двух матриц 6

2 Конструкторский раздел **8**

2.1 Разработка алгоритма поиска расстояния Левенштейна 8

2.2 Разработка алгоритма поиска расстояния Дамерау- Левенштейна 9

3 Технологический раздел **13**

3.1 Средства реализации 13

3.2 Сведения о модулях программы 13

3.3 Реализации алгоритмов 14

3.4 Функциональные тесты 17

4 Исследовательский раздел **19**

4.1 Технические характеристики 19

4.2 Время выполнения алгоритмов 19

4.3 Использование памяти 21

ЗАКЛЮЧЕНИЕ **26**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **27**

ВВЕДЕНИЕ

Матрица в математике – таблица чисел, состоящая из определенного количества строк и столбцов.

Умножение матриц – одна из основных операций над матрицами. Оно используется в различных областях, включая машинное обучение, обработку изображений, и многие другие.

Целью данной лабораторной работы является изучение, описание, реализация и исследование классического алгоритма умножения матриц, умножения матриц с использованием алгоритма Винограда, а также с использованием его оптимизированной версии согласно варианту.

Для достижения поставленной цели необходимо решить следующие задачи:

1) Изучить и описать алгоритмы классического умножения матриц и умножения матриц с использованием алгоритма Винограда.

2) Создать программное обеспечение, реализующее следующие алгоритмы:

   – классический алгоритм умножения матриц;

   – умножение матриц с использованием алгоритма Винограда;

   – умножение матриц с использованием оптимизированной версии алгоритма Винограда.

3) Провести анализ эффективности реализаций алгоритмов по памяти и по времени.

4) Провести оценку сложности алгоритмов и казать влияние оптимизаций на характеристики программной реализации.

5) Обосновать полученные результаты в отчете к выполненной лабораторной работе.

1 Аналитический раздел

В данном разделе будут рассмотрены понятия матрицы, умножения двух матриц, классический алгоритм умножения матриц и умножение матриц с помощью алгоритма Винограда.

1.1 Матрица

Матрица – математический объект, записываемый в виде прямоугольной таблицы элементов кольца или поля (например, целых или комплексных чисел), которая представляет собой совокупность строк и столбцов, на пересечении которых находятся её элементы. Количество строк и столбцов матрицы задают размер матрицы [1].

$$A_{m\times n} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$

Для матрицы определены следующие алгебраические операции:

1) Сложение матриц, имеющих один и тот же размер.

2) Умножение матрицы на число.

3) Умножение матриц подходящего размера.

Умножение двух матриц (обозначается: AB, реже $A \times B$) определяется следующим образом: каждый элемент результирующей матрицы – это сумма произведений элементов соответствующих строк первой матрицы и столбца второй матрицы. При этом количество столбцов в первой матрице должно совпадать с количеством строк во второй матрице. Операция умножения матриц в общем случае не коммутативна, то есть $AB \neq BA$.

1.2 Классический алгоритм умножения двух матриц

Классический алгоритм умножение двух матриц вытекает из определения умножения двух матриц и реализует формулу 1.1.

Пусть даны две прямоугольные матрицы A и B размерности $m \times n$ и $n \times q$ соответственно:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{l1} & a_{l2} & \cdots & a_{lm} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1n} \\ b_{21} & b_{22} & \cdots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \cdots & b_{mn} \end{pmatrix}.$$

Тогда матрица C размерностью $m \times q$:

$$C = \begin{pmatrix} c_{11} & c_{12} & \cdots & c_{1q} \\ c_{21} & c_{22} & \cdots & c_{2q} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mq} \end{pmatrix}$$

где элемент результирующей матрицы c_{ij} определяется так:

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj} \tag{1.1}$$

1.3 Алгоритм Винограда для умножения двух матриц

Алгоритм Винограда [2] – алгоритм умножения квадратных матриц. Анализируя классический алгоритм умножения двух матриц, можно увидеть, что каждый элемент результирующей матрицы представляет собой скалярное произведение соответствующей строки и соответствующего столбца исходной матрицы.

Рассмотрим 2 вектора: $V = (v_1, v_2, v_3, v_4)$ и $W = (w_1, w_2, w_3, w_4)$.

Их скалярное произведение равно:

$$V \cdot W = v_1 \cdot w_1 + v_2 \cdot w_2 + v_3 \cdot w_3 + v_3 \cdot w_3 \tag{1.2}$$

Это равенство можно переписать в виде:

$$V \cdot W = (v_1 + w_2) \cdot (v_2 + w_1) + (v_3 + w_4) \cdot (v_4 + w_3) \\ -v_1 \cdot v_2 - v_3 \cdot v_4 - w_1 \cdot w_2 - w_3 \cdot w_4 \tag{1.3}$$

Несмотря на то, что второе выражение требует вычисления большего количества операций, чем стандартный алгоритм: вместо четырех умножений - шесть, а вместо трех сложений - десять, последние слагаемые в формуле 1.3 допускают предварительную обработку: его части можно вычислить заранее и запомнить для каждой строки первой матрицы и для каждого столбца второй матрицы, что позволит для каждого элемента выполнять лишь два умножения и пять сложений, складывая затем только лишь с 2 предварительно посчитанными суммами соседних элементов текущих строк и столбцов. Из-за того, что операция сложения быстрее операции умножения в ЭВМ, на практике алгоритм должен работать быстрее стандартного.

В случае нечетного значений размера изначальной матрицы следует произвести еще одну операцию - добавление произведения последних элементов соответствующих строк и столбцов.

1.4 Оптимизированный алгоритм Винограда для умножения двух матриц

При программной реализации алгоритма Винограда предлагается выполнить следующие оптимизации:

1) Заменить умножение на 2 на побитовый сдвиг влево.

2) Заменить выражение вида $x = x + k$ на выражение вида $x += k$.

3) Значение $\frac{Q}{2}$, используемое в циклах расчета предварительных данных, вычислить заранее.

Вывод

В данном разделе были рассмотрены понятия матрицы и операции умножения, классического алгоритма умножения матриц и алгоритма умножения матриц с помощью алгоритма Винограда, а также были приведены варинты оптимизаций алгоритма Винограда.

2 Конструкторский раздел

В данном разделе будут разработаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна и приведены схемы алгоритмов различных способов их реализации.

2.1 Разработка алгоритма поиска расстояния Левенштейна

На рисунках ?? и ?? приведена схема нерекурсивной реализации алгоритма нахождения расстояния Левенштейна.

2.2 Разработка алгоритма поиска расстояния Дамерау-Левенштейна

На рисунках ?? и ?? приведена схема нерекурсивной реализации алгоритма нахождения расстояния Дамерау-Левенштейна.

На рисунке **??** приведена схема рекурсивной реализации алгоритма нахождения расстояния Дамерау-Левенштейна.

На рисунке **??** приведена схема рекурсивной реализации алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием.

Вывод

В данном разделе были перечислены основные требования к программному обеспечению, а также, на основе данных, полученных в аналитическом разделе, были построены схемы алгоритмов различных способов реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

3 Технологический раздел

В данном разделе будут перечислены средства реализации, листинги кода и функциональные тесты.

3.1 Средства реализации

В качестве языка программирования для этой лабораторной работы был выбран $C++$ [3] по следующим причинам:

- в $C++$ есть встроенный модуль *ctime*, предоставляющий необходимый функционал для замеров процессорного времени;
- в стандартной библиотеке $C++$ есть оператор $sizeof$, позволяющий получить размер переданного объекта в байтах. Следовательно, $C++$ предоставляет возможности для проведения точных оценок по используемой памяти;
- в $C++$ есть тип данных $std :: wstring$, который позволяет хранить и использовать как кириллические, так и латинские символы.

В качестве функции, которая будет осуществлять замеры процессорного времени, будет использована функция $clock\_gettime$ из встроенного модуля *ctime* [4].

3.2 Сведения о модулях программы

Программа состоит из семи модулей:

1) `algorithms.cpp` — модуль, хранящий реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;
2) `processTime.cpp` — модуль, содержащий функцию для замера процессорного времени;
3) `memoryMeasurements.cpp` — модуль, содержащий функции, позволяющие провести сравнительный анализ использования памяти в реализациях алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;

4) `timeMeasurements.cpp` — модуль, содержащий функции, позволяющие провести сравнительный анализ использования времени в реализациях алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна;

5) `matrix.cpp` — модуль, содержащий набор функций для работы с Си-матрицей.

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

7) `task7` — модуль, содержащий набор скриптов для проведения замеров программы по времени и памяти и построения графиков по полученным данным.

3.3 Реализации алгоритмов

В листингах 3.1, 3.2, 3.3 приведены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Реализация классического алгоритма умножения двух матриц

```
int **matrixMulClassic(const int *const *mtr1,
                       const int *const *mtr2,
                       int cRow, int cCol, int cColRes)
{
    int **mtrRes = matrix_alloc(cRow, cColRes, 0);

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cColRes; ++j)
            for (int k = 0; k < cCol; ++k)
                mtrRes[i][j] = mtrRes[i][j] + mtr1[i][k] *
                   mtr2[k][j];

    return mtrRes;
}
```

Листинг 3.2 – Реализация алгоритма Винограда для умножения двух матриц

```
int **matrixMulVinograd(const int *const *const mtr1,
                        const int *const *mtr2,
                        int cRow, int cCol, int cColRes)
{
    int **mtrRes = matrix_alloc(cRow, cColRes, 0);
    int rowFactor[cRow], colFactor[cColRes];

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cCol / 2; ++j)
            rowFactor[i] = rowFactor[i] + mtr1[i][2 * j] *
               mtr1[i][2 * j + 1];

    for (int j = 0; j < cColRes; ++j)
        for (int i = 0; i < cCol / 2; ++i)
            colFactor[j] = colFactor[j] + mtr2[i * 2][j] * mtr2[i
               * 2 + 1][j];

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cColRes; ++j)
        {
            mtrRes[i][j] = -rowFactor[i] - colFactor[j];

            for (int k = 0; k < cCol / 2; ++k)
                mtrRes[i][j] = mtrRes[i][j] + (mtr1[i][2 * k] +
                   mtr2[2 * k + 1][j]) * (mtr1[i][2 * k + 1] +
                   mtr2[2 * k][j]);
        }

    if (cCol % 2 != 0)
        for (int i = 0; i < cCol; ++i)
            for (int j = 0; j < cColRes; ++j)
                mtrRes[i][j] = mtrRes[i][j] + mtr1[i][cCol - 1] *
                   mtr2[cCol - 1][j];

    return mtrRes;
}
```

Листинг 3.3 – Реализация оптимизированного алгоритма Винограда для умножения двух матриц

```
int **matrixMulVinogradWithOpt(const int *const *const mtr1,
                               const int *const *mtr2,
                               int cRow, int cCol, int cColRes)
{
    int **mtrRes = matrix_alloc(cRow, cColRes, 0);
    int rowFactor[cRow], colFactor[cColRes];
    int cCol_half = cCol / 2;

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cCol_half; ++j)
            rowFactor[i] += mtr1[i][j << 1] * mtr1[i][(j << 1) +
               1];

    for (int j = 0; j < cColRes; ++j)
        for (int i = 0; i < cCol_half; ++i)
            colFactor[j] += mtr2[i << 1][j] * mtr2[(i << 1) +
               1][j];

    for (int i = 0; i < cRow; ++i)
        for (int j = 0; j < cColRes; ++j)
        {
            mtrRes[i][j] = -rowFactor[i] - colFactor[j];
            for (int k = 0; k < cCol_half; ++k)
                mtrRes[i][j] += (mtr1[i][k << 1] + mtr2[(k << 1) +
                   1][j]) * (mtr1[i][(k << 1) + 1] + mtr2[k <<
                   1][j]);
        }

    if (cCol % 2 != 0)
        for (int i = 0; i < cCol; ++i)
            for (int j = 0; j < cColRes; ++j)
                mtrRes[i][j] += mtr1[i][cCol - 1] * mtr2[cCol -
                   1][j];

    return mtrRes;
}
```

3.4 Функциональные тесты

В таблице 3.1 и 3.2 приведены функциональные тесты для разработанных алгоритмов умножения матриц. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты для классического алгоритма умножения матриц

Входные данные		**Результат для классического алгоритма**	
Матрица 1	**Матрица 2**	**Ожидаемый результат**	**Фактический результат**
$\begin{pmatrix} 1 & 5 & 7 \\ 2 & 6 & 8 \\ 3 & 7 & 9 \end{pmatrix}$	$\begin{pmatrix} \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 1 & 5 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
$\begin{pmatrix} 3 & 5 \\ 2 & 1 \\ 9 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	$\begin{pmatrix} 23 & 31 & 39 \\ 6 & 9 & 12 \end{pmatrix}$	$\begin{pmatrix} 23 & 31 & 39 \\ 6 & 9 & 12 \end{pmatrix}$
$\begin{pmatrix} 10 \end{pmatrix}$	$\begin{pmatrix} 35 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$

Таблица 3.2 – Функциональные тесты для умножения матриц по алгоритму Винограда

Входные данные		**Результат для алгоритма Винограда**	
Матрица 1	**Матрица 2**	**Ожидаемый результат**	**Фактический результат**
$\begin{pmatrix} 1 & 5 & 7 \\ 2 & 6 & 8 \\ 3 & 7 & 9 \end{pmatrix}$	$\begin{pmatrix} \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 1 & 5 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$
$\begin{pmatrix} 3 & 5 \\ 2 & 1 \\ 9 & 7 \end{pmatrix}$	$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$	Сообщение об ошибке	Сообщение об ошибке
$\begin{pmatrix} 10 \end{pmatrix}$	$\begin{pmatrix} 35 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$	$\begin{pmatrix} 350 \end{pmatrix}$

Вывод

В данном разделе были реализованы и протестированы три алгоритма: классический алгоритм умножения, алгоритм Винограда умножения двух матриц, оптимизированный алгоритм Винограда умножения двух матриц.

4 Исследовательский раздел

В данном разделе будут проведены сравнения реализаций алгоритмов умножения матриц по времени работы и по затрачиваемой памяти.

4.1 Технические характеристики

Технические характеристики устройства, на котором проводились исследования:

– операционная система: Ubuntu 22.04.3 LTS x86_64 [5];

– оперативная память: 16 Гб;

– процессор: 11th Gen Intel® Core™ i7-1185G7 @ 3.00GHz × 8.

4.2 Время выполнения алгоритмов

Время работы алгоритмов измерялось с использованием функции *clock_gettime* из встроенного модуля *ctime*.

Замеры времени для каждого размера матрицы проводились 1000 раз. На вход подавались случайно сгенерированные матрицы заданного размера.

Результаты замеров реализаций алгоритмов по времени представлены в таблице 4.1. Их графическое представление показано на рисунке 4.1.

Таблица 4.1 – Замер памяти для матриц размером от 10 до 100

Линейный размер, штуки	**Время, нс**		
	Классический	**Виноград**	**Виноград (опт)**
10	3416.896	3180.420	3134.480
20	29545.140	22744.160	21871.700
30	94943.940	74144.360	73816.280
40	218554.000	177503.000	169667.400
50	433830.200	345092.400	332352.600
60	742386.600	588659.800	574165.200
70	1150328.000	930386.000	915796.200
80	1718110.000	1381020.000	1344386.000
90	2463566.000	1984904.000	1919098.000
100	3386088.000	2715484.000	2632198.000

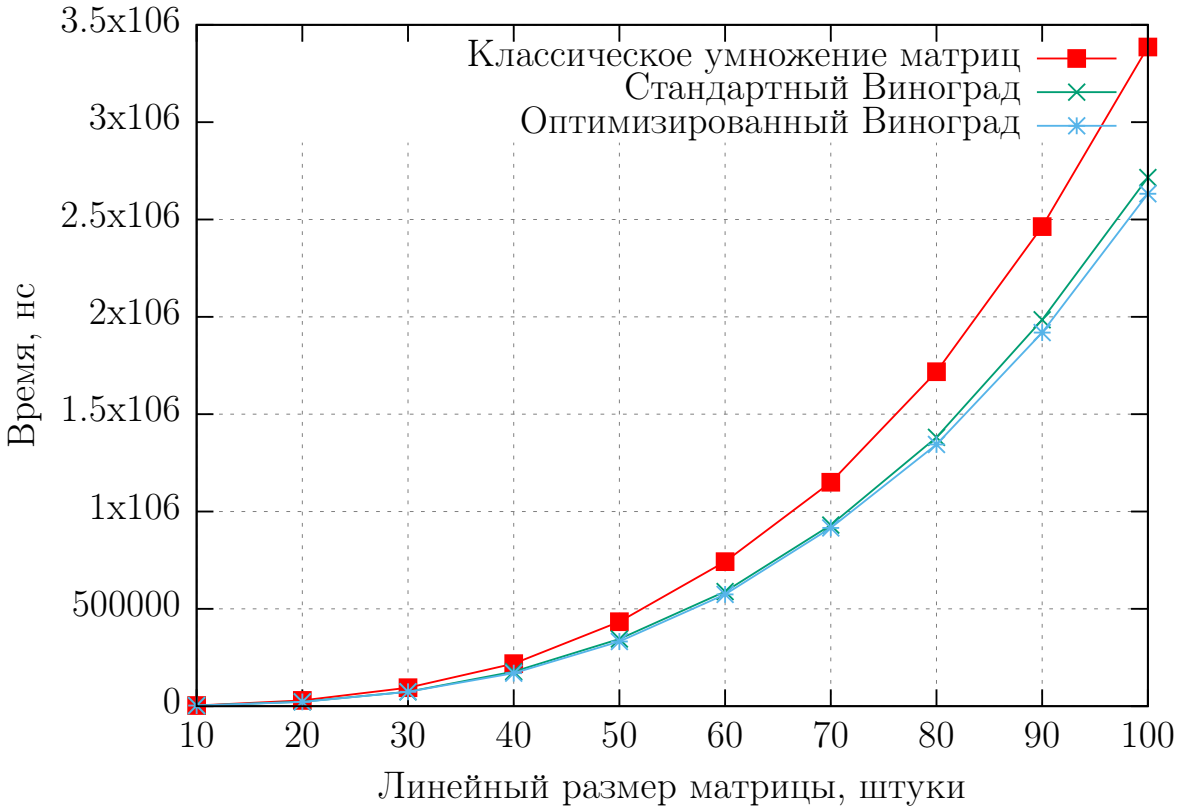

Рисунок 4.1 – Результаты замеров времени работы алгоритмов для матриц размеров от 10 до 100

4.3 Использование памяти

Введем следующие обозначения:

– l_1 — длина строки S_1;

– l_2 — длина строки S_2;

– $sizeof()$ — функция вычисляющая размер в байтах;

– $wstring$ — строковый тип;

– int — целочисленный тип;

– $size\_t$ — беззнаковый целочисленный тип.

Максимальная глубина стека вызовов при рекурсивной реализации нахождения расстояния Дамерау-Левенштейна равна сумме входящих строк, а на каждый вызов требуется 2 дополнительные переменные, соответственно, максимальный расход памяти равен:

$$(l_1 + l_2) \cdot (2 \cdot sizeof(size\_t) + 2 \cdot sizeof(int)) + 2 * sizeof(wstring), \qquad (4.1)$$

где:

– $2 \cdot sizeof(size\_t)$ — хранение размеров строк;

– $2 \cdot sizeof(int)$ — дополнительные переменные;

– $2 \cdot sizeof(wstring)$ — хранение двух строк.

Расчет используемой памяти для рекурсивного алгоритма с кешированием поиска расстояния Дамерау-Левенштейна будет теоретически схож с расчетом в формуле 4.1, но также учитывается матрица, соответственно, максимальный расход памяти равен:

$$(l_1 + 1) \cdot (l_2 + 1) \cdot sizeof(int) + sizeof(int**) + (l_1 + 1) \cdot sizeof(int*) + (l_1 + l_2) \cdot (2 \cdot sizeof(size\_t) + sizeof(int)) + 2 \cdot sizeof(wstring) \quad (4.2)$$

где

– $(l_1 + 1) \cdot (l_2 + 1) \cdot sizeof(int)$ — хранение матрицы;

– $sizeof(int**)$ – указатель на матрицу;

– $(l_1 + 1) \cdot sizeof(int*)$ – указатель на строки матрицы;

– $2 \cdot size(size\_t)$ — хранение размеров строк;

– $sizeof(int)$ — дополнительная переменная;

– $2 * sizeof(wstring)$ — хранение двух строк.

Расчет использования памяти при итеративной реализации алгоритма поиска расстояния Левенштейна теоретически равен:

$$2 \cdot (l_2 + 1) \cdot sizeof(int) + 2 \cdot sizeof(wstring) + 2 \cdot sizeof(size\_t) + sizeof(int), \quad (4.3)$$

где

– $2 \cdot (l_2 + 1) \cdot sizeof(int)$ — хранение двух массивов;

– $2 \cdot size(wstring)$ — хранение двух строк;

– $2 \cdot sizeof(size\_t)$ — хранение размеров строк;

– $sizeof(int)$ — дополнительная переменная.

Расчет использования памяти при итеративной реализации алгоритма поиска расстояния Дамерау-Левенштейна теоретически равен:

$$(l_1 + 1) \cdot (l_2 + 1) \cdot sizeof(int) + sizeof(int**) + (l_1 + 1) \cdot sizeof(int*) + 2 \cdot sizeof(wstring) + 2 \cdot sizeof(size\_t) + sizeof(int), \quad (4.4)$$

где

– $(l_1 + 1) \cdot (l_2 + 1) \cdot sizeof(int)$ — хранение матрицы;

– $sizeof(int **)$ – указатель на матрицу;

– $(l_1 + 1) \cdot sizeof(int*)$ – указатель на строки матрицы;

– $2 * sizeof(wstring)$ — хранение двух строк;

– $2 \cdot size(size\_t)$ — хранение размеров матрицы;

– $sizeof(int)$ — дополнительная переменная.

Таблица 4.2 – Замер памяти для строк, размером от 10 до 200

Длина, символ	**Размер, байты**			
	Левенштейн	**Дамерау-Левенштейн**		
	Итеративный	**Итеративный**	**Рекурсивный**	
			Без кеша	**С кешом**
10	252	748	624	1128
20	412	2188	1184	2968
30	572	4428	1744	5608
40	732	7468	2304	9048
50	892	11308	2864	13288
60	1052	15948	3424	18328
70	1212	21388	3984	24168
80	1372	27628	4544	30808
90	1532	34668	5104	38248
100	1692	42508	5664	46488

Анализируя таблицу 4.2, сравним нерекурсивные реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна. При любой длине строки от 10 до 100 символов итеративный алгоритм поиска расстояния Левенштейна использует меньше памяти: при длине строки в 10 символов - в 3 раза меньше, а при длине строки в 100 символов уже в 25 раз. Такие результаты объясняются тем, что нерекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна использует матрицу для сохранения ранее вычисленных значений, в то время как нерекурсивной реализации алгоритма поиска расстояния Левенштейна необходимы лишь текущая и предыдущая строки матрицы.

Сравнивая рекурсивную и рекурсивную с кешированием реализации алгоритмов поиска расстояний Левенштейна, можно увидеть, что использования матрицы в качестве кеша также приводит к быстрому росту используемой памяти в зависимости от длины входных строк.

При рассмотрении нерекурсивной и рекурсивной реализаций алгоритма поиска расстояний Дамерау-Левенштейна видно, что последняя с ростом длины строки использует в несколько раз меньше памяти, чем нерекурсивная: при длине строки в 10 символов – в 1.2 раза меньше, а при длине строки в 100 символов – в 7.5 раз меньше. Это связано с тем, что нерекурсивная реализация использует матрицу, в то время как рекурсивная использует только память, выделенную под локальные переменные при каждом рекурсивном вызове функции.

Вывод

В данном разделе были проведены замеры времени работы а также расчеты используемой памяти реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Итеративные реализации алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна работают быстрее рекурсивных, поскольку при итеративных реализациях не происходит многократного расчета одних и тех же промежуточных значений в ходе работы алгоритма.

Однако, рекурсивные алгоритмы более эффективные при использовании памяти, поскольку при использовании рекурсивной реализации происходит выделение памяти только под локальные переменные при каждом рекурсивном

вызове.

Использование матрицы в качестве кеша в рекурсивной реализации алгоритма Дамерау-Левенштейна позволило сократить время работы алгоритма, но увеличило количество используемой памяти.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были решены следующие задачи:

1) Изучены и описаны алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна;

2) Создано программное обеспечение, реализующее следующие алгоритмы:

   — нерекурсивный алгоритм поиска расстояния Левенштейна;

   — нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна с кешированием.

3) Проведены анализы эффективности реализаций алгоритмов по памяти и по времени;

4) Подготовлен отчет о лабораторной работе.

Цель данной лабораторной работы, а именно изучение, описание, реализация и исследование алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна, также была достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Матрица, её история и применение [Электронный ресурс]. — Режим доступа: https://urok.1sept.ru/articles/637896 (дата обращения: 11.10.2022).
2. Реализация алгоритма умножения матриц по винограду на языке Haskell [Электронный ресурс]. — Режим доступа: https://cyberleninka.ru/article/n/realizatsiya-algoritma-umnozheniya-matrits-po-vinogradu-na-yazyke-haskell (дата обращения: 11.10.2022).
3. Справочник по языку C++ [Электронный ресурс]. — Режим доступа: https://learn.microsoft.com/ru-ru/cpp/cpp/cpp-language-reference?view=msvc-170 (дата обращения: 28.09.2022).
4. clock_getres [Электронный ресурс]. — Режим доступа: https://pubs.opengroup.org/onlinepubs/9699919799/functions/clock_getres.html (дата обращения: 28.09.2022).
5. Ubuntu 22.04.3 LTS (Jammy Jellyfish) [Электронный ресурс]. — Режим доступа: https://releases.ubuntu.com/22.04/ (дата обращения: 28.09.2022).