Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н. Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Р А С Ч Ё Т Н О - П О Я С Н И Т Е Л Ь Н А Я З А П И С К А

к лабораторной работе на тему:

Перемножение матриц : алгоритм Винограда

Студент _____ Киселев А.М.
(Подпись, дата)

Преподаватель _____ Волкова Л.Л.
(Подпись, дата)

Москва 2018

Содержание

Введение

Целью работы является изучение нахождения трудоемкости на материале алгоритма перемножения матриц Винограда. Для достижения поставленной цели необходимо решить следующие задачи:

— Изучить алгоритм Винограда;

— Получить улучшенную версию Винограда и сравнить ее с обычной реализацией;

— Провести оценку трудоемкости алгоритма;

— Рассмотреть лучший и худший случаи при перемножении матриц данным алгоритмом;

— Привести эксперементальное подтверждение различий во временной эффективности оптимизированного алгоритма и обычного при помощи разработанного ПО на материале замеров процессорного времени выполнения реализации на варьирующихся размерах перемножаемых матриц;

— Описать и обосновать полученные результаты о выполненной работе;

1 Аналитический раздел

Умножение матриц – важная задача в разных областях, которая помогает рассчитать системы линейных уравнение, которые в свою очередь применяются в системах моделирования реального мира, 3D моделировании, в обработке и хранении информации.

Задача по эффективной обработке матриц является актуальной и востребованной в наши дни. Первым шагом на пути к более быстрому по времени перемножении матриц стал алгоритм Винограда. Он представляет собой перемножение матриц с использованием скалярного произведения соответствующих строки и столбца исходных матриц.

Более подробное описание алгоритма:

1.1 Алгоритм Винограда

Пусть даны матрицы A[N*M] и B[M*K], тогда их результирующее произведение можно представить как матрицу C[N*K]. При умножении "в лоб"мы получаем много операций умножения, которые являются очень трудоемкими по сравнению со сложением:

$$A * B = v_1 * w_1 + v_2 * w_2 + v_3 * w_3 + ... + v_m * w_m \tag{1.1}$$

Но это выражение можно представить в другом виде, в котором хорошо будут видны элементы, которые возможно рассчитать заранее и использовать несколько раз. Рассмотрим на частном случае: Пусть даны вектора $V = (v1, v2, v3, v4)$ и $W = (w1, w2, w3, w4)$. Произведение $V * W$ можно найти по формуле 1.1, которая представлена в общем виде, но можно записать так:

$$V*W = (v_1+w_2)*(v_2+w_1)+(v_3+w_4)*(v_4+w_3)-v_1*v_2-v_3*v_4-w_1*w_2-w_3*w_4 \tag{1.2}$$

Часть $-v_1*v_2-v_3*v_4-w_1*w_2-w_3*w_4$ из выражения 1.2 вычислима заранее, что позволят нам избавиться от лишних трудоемких операций умножения.

1.2 Рассчет трудоемкости алгоритма

Рассчет трудоемкости алгоритма опирается на модель вычисления, в которой нужно определить свод правил : оценка операций, циклов, стоимость перехода по условиям.

2 Констукторский раздел

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

2.1 Разработка алгоритмов

2.1.1 Расстояние Левенштейна(обычный)

На рисунке 2.1 представлена блок схема алгоритма Левенштейна.

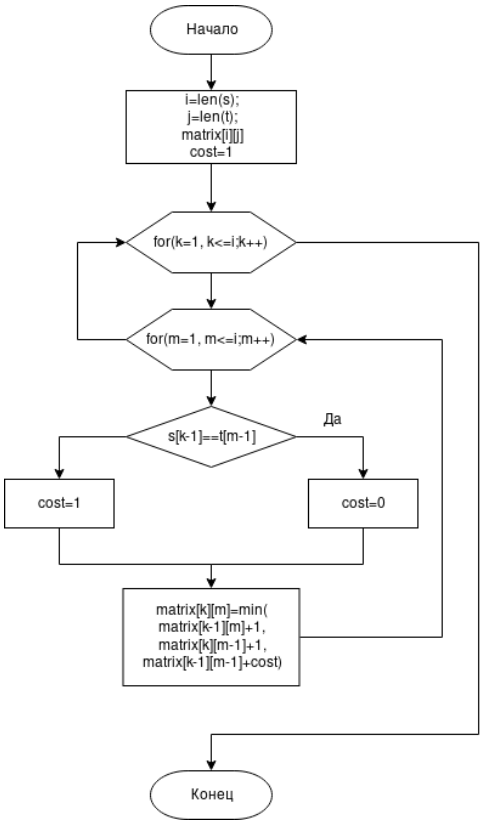

Рисунок 2.1 — Представлена схема алгоритма нахождения расстояния Левенштейна для матричной реализации

2.1.2 Расстояние Дамерау – Левенштейна

На рисунке 2.2 представлена блок схема алгоритма Дамерау – Левенштейна.

2.1.3 Расстояние Левенштейна(рекурсивный)

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

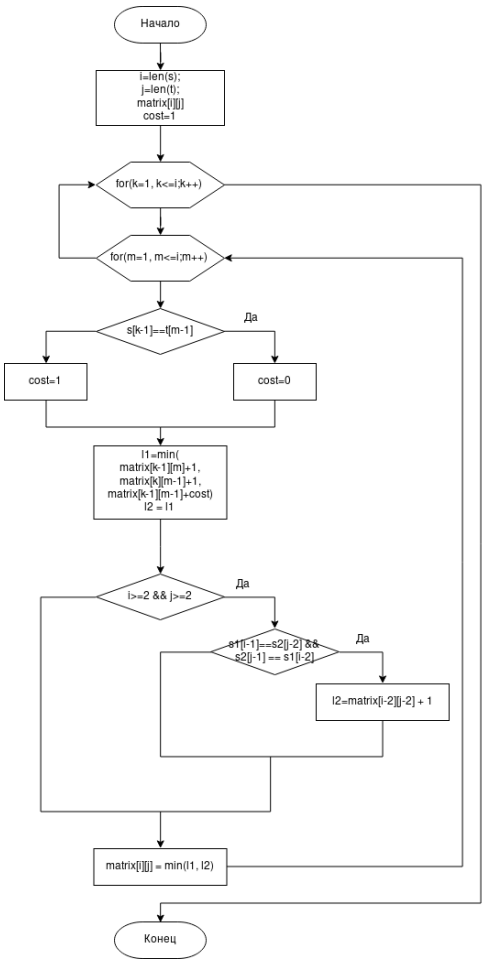

Рисунок 2.2 — Представлена схема алгоритма нахождения расстояния Дамерау – Левенштейна для матричной реализации

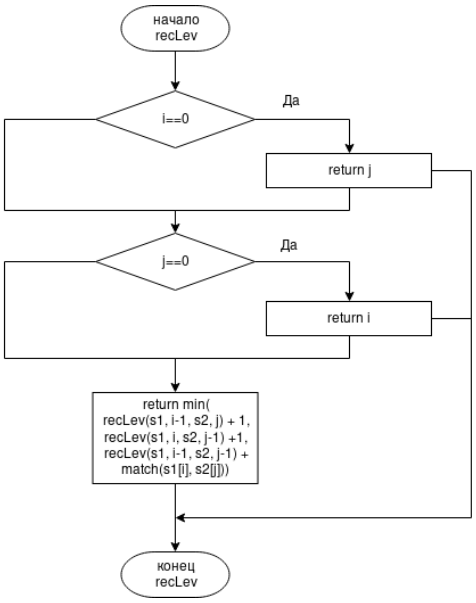

Рисунок 2.3 — Представлена схема алгоритма нахождения расстояния Левенштайна для рекурсивной реализации

3 Технологический раздел

Данный раздел содержит информацию о реализации ПО и листингах программ.

3.1 Требования к программному ПО

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

3.2 Средства реализации

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

3.3 Листинг

Алгоритм Левенштейна представлен в листинге 3.1

```
int levensteinDistance( char* s1, char* s2,
                        int ( * matrix)[MAX_STRING_SIZE][MAX_STRING_SIZE] )
{
    size_t len1 = strlen(s1);
    size_t len2 = strlen(s2);

    matrixInit(matrix, len1, len2);

    int cost = 1;
    for(int i = 1; i <= len1; i++)
        for(int j = 1; j <= len2; j++) {
            if(s1[i-1] == s2[j-1])
                cost = 0;
            else
                cost = 1;
            ( * matrix)[i][j] = min( 3,
                        ( * matrix)[i-1][j]+1,
                        ( * matrix)[i][j-1]+1,
                        ( * matrix)[i-1][j-1]+cost );
        }
}
```

Листинг 3.1 — Levenstein algorithm

Листинг алгоритма Дамерау – Левенштейна представлен в листинге 3.2

```
int levensteinDistance( char* s1, char* s2,
                        int ( * matrix)[MAX_STRING_SIZE][MAX_STRING_SIZE] )
{
    size_t len1 = strlen(s1);
    size_t len2 = strlen(s2);

    matrixInit(matrix, len1, len2);

    int cost = 1;
    for(int i = 1; i <= len1; i++)
        for(int j = 1; j <= len2; j++) {
            int flag = 0;
            if(s1[i-1] == s2[j-1])
                cost = 0;
            else
                cost = 1;
            int l1= min( 3,
                        ( * matrix)[i-1][j]+1,
                        ( * matrix)[i][j-1]+1,
                        ( * matrix)[i-1][j-1]+cost );
            int l2 = l1;
            if(i>=2 && j>=2)
                if(s1[i-1] == s2[j-2] && s2[j-1] == s1[i-2])
                    l2 = ( * matrix)[i-2][j-2]+1;
            ( * matrix)[i][j] = min(2, l1, l2);
        }
}
```

Листинг 3.2 — Damerau – Levenstein algorithm

Листинг алгоримта Левенштейна в рекурсивной реализации представлен в листинге 3.3

```
int levensteinDistance( char* s1, int i, char* s2, int j )
{
    if(i == 0) return j;
    if(j == 0) return i;

    return min( 3,levensteinDistance(s1, i-1, s2, j) + 1,
                  levensteinDistance(s1, i, s2, j-1) + 1,
                  levensteinDistance(s1, i-1, s2, j-1) + match(s1[i], s2[j
    ]) );
}
```

Листинг 3.3 — Recursive Levenstein algorithm

Листинг функции match() представлен в листинге 3.4

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

Листинг 3.4 — Function match()

4 Экспериментальный раздел

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

4.1 Примеры работы

Алгоритм Левенштейна:

```
Input s1: OTAPA
Input s2: TAPTAP

0 1 2 3 4 5 6
1 1 2 3 4 5 6
2 1 2 3 3 4 5
3 2 1 2 3 3 4
4 3 2 1 2 3 3
5 4 3 2 2 2 3
```

```
Input s1: MGTU
Input s2: MTGU

0 1 2 3 4
1 0 1 2 3
2 1 1 1 2
3 2 1 2 2
4 3 2 2 2
```

Алгоритм Дамерау – Левенштейна:

```
Input s1: MGTU
Input s2: MTGU

0 1 2 3 4
1 0 1 2 3
2 1 1 1 2
3 2 1 1 2
4 3 2 2 1
```

```
Input s1: KNUTH
Input s2: TANENBAUM

0 1 2 3 4 5 6 7 8 9
```

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

Алгоритм Левенштейна(Рекурсивный):

```
1 | Input s1: MGTU
2 | Input s2: MTGU
3 | Length is : 2
```

```
1 | Input s1: OTAPA
2 | Input s2: TAPTAP
3 | Length is : 3
```

4.2 Постановка эксперимента

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

4.3 Сравнительный анализ на материале экспериментальных данных

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

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

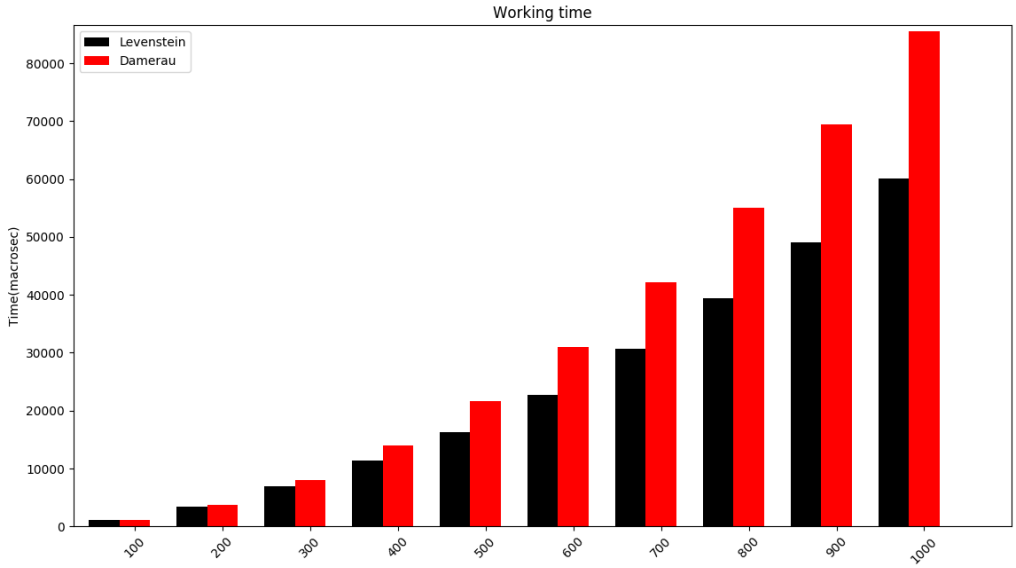

Рисунок 4.1 — Эксперемент первый, демонстрация различия времени для алгоритмов Левенштейна и Дамерау – Левенштейна

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

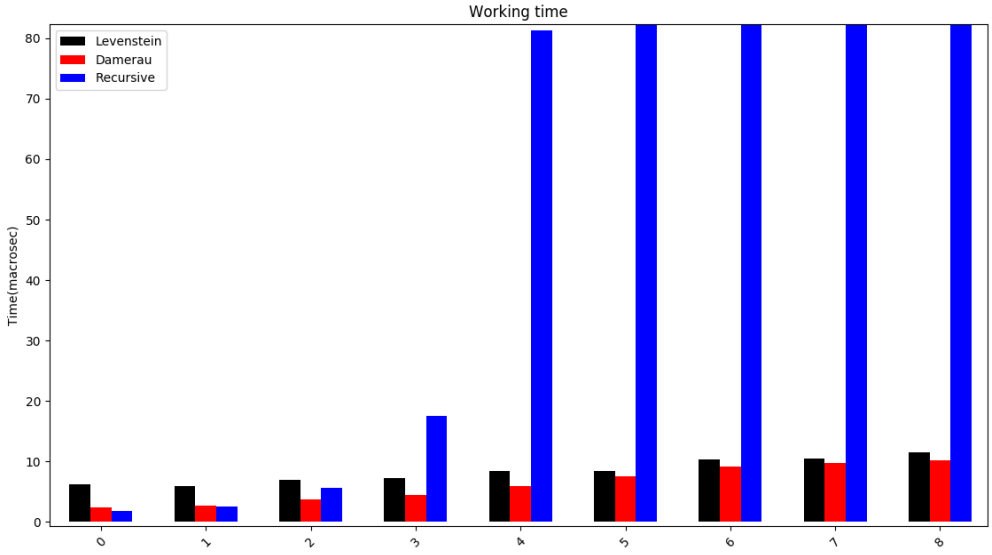

Рисунок 4.2 — Эксперемент второй, демонстрация различия времени для алгоритмов Левенштейна и Дамерау – Левенштейна и рекурсивной реализации Левенштейна

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

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

```
 Input s1: FLEICHE
 Input s2: EUROPA
Length is : 6
function called 29737 times
```

```
 Input s1: ENTSCHULDIGUNG
 Input s2: EINWAGEN
Length is : 11
function called 21820057 times
```

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

```
function called 94 times
(MGT, MTG)
 `--(MG, MT)
  |    `--(M, M)
  |    |    `--(, )
  |    |    `--(M, )
  |    |    `--(, M)
  |    `--(MG, M)
  |    |    `--(M, )
  |    |    `--(MG, )
  |    |    `--(M, M)
  |    |         `--(, )
  |    |         `--(M, )
  |    |         `--(, M)
  |    `--(M, MT)
  |         `--(, M)
  |         `--(M, M)
  |         |    `--(, )
  |         |    `--(M, )
  |         |    `--(, M)
  |         `--(, MT)
```

```
‘−−(MGT, MT)
|   ‘−−(MG, M)
|   |   ‘−−(M, )
|   |   ‘−−(MG, )
|   |   ‘−−(M, M)
|   |       ‘−−(, )
|   |       ‘−−(M, )
|   |       ‘−−(, M)
|   ‘−−(MGT, M)
|   |   ‘−−(MG, )
|   |   ‘−−(MGT, )
|   |   ‘−−(MG, M)
|   |       ‘−−(M, )
|   |       ‘−−(MG, )
|   |       ‘−−(M, M)
|   |           ‘−−(, )
|   |           ‘−−(M, )
|   |           ‘−−(, M)
|   ‘−−(MG, MT)
|       ‘−−(M, M)
|       |   ‘−−(, )
|       |   ‘−−(M, )
|       |   ‘−−(, M)
|       ‘−−(MG, M)
|       |   ‘−−(M, )
|       |   ‘−−(MG, )
|       |   ‘−−(M, M)
|       |       ‘−−(, )
|       |       ‘−−(M, )
|       |       ‘−−(, M)
|       ‘−−(M, MT)
|           ‘−−(, M)
|           ‘−−(M, M)
|           |   ‘−−(, )
|           |   ‘−−(M, )
|           |   ‘−−(, M)
|           ‘−−(, MT)
‘−−(MG, MTG)
    ‘−−(M, MT)
    |   ‘−−(, M)
    |   ‘−−(M, M)
    |   |   ‘−−(, )
    |   |   ‘−−(M, )
    |   |   ‘−−(, M)
    |   ‘−−(, MT)
    ‘−−(MG, MT)
    |   ‘−−(M, M)
```

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

Листинг 4.1 — Recursive tree example MGT, MTG

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

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

Рекурсивная реализация алгоритма Левенштейна(valgrind):

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

Алгоритм Левенштйена(valgrind):

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

Алгоритм Дамерау – Левенштейна(valgrind):

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

Заключение

В результате выполнения задания были получены следующие основные результаты:

— изучены теоретические понятия редукционного расстояиния

— изучены основные алгоритмы нахождение редукционного расстояния между строками: алгоритм Левенштейна, Дамерау – Левенштейна

— проведен аналитичесчкий вывод формул для заполнения матриц расстояний

— проведено сравнение трех реализаций заданных алгоритмов, были выявлены их слабые мест

— в рамках данной работы было выяснено, что рекурсивный алгоритм работает намного медленнее двух других из-за большого количества рекурсивных вызовов, что делает его бесполезным при работе с большими объемами информации, а алгоритм Левенштейна работает медленнее Дамерау – Левенштейна на маленьких словах, но на словах больше 6 символов, Дамерау – Левенштейн начинает работать медленнее