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:

Рекурентные соотношения: расстояние между строками

Студент ______________________________ Киселев А.М.

(Подипсь, дата)

Преподаватель ______________________________ Волковаю Л.Л.

(Подпись, дата)

Москва 2018

Содержание

Введение

Целью работы является изучение и применение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна, а так же реализовать алгоритм Левенштейна в рекурсивном виде. Для достижения поставленной цели необходимо решить следующие задачи:

— Изучить алгоритмы Левенштейна и Дамера-Левенштейна нахождения расстояния между строками;

— Применить метод динамического програмирования для матричной реализации указаных алгоритмов;

— Получить практические навыки реализации указанных алгоритмов: двух алгоритмов в матричной версии и алгоритма Левенштейна, реализованного рекурсивно;

— Провести сравнительный анализ линейной и рекурсивной реализаций алгоритма Левенштейна по затрачиваемым ресурсам(времени и памяти);

— Привести эксперементальное подтверждение различий во временной эффективности рекурсивной и нерекурсивной реализаций алгоритма Левенштейна при помощи разработанного ПО на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;

— Описать и обосновать полученные результаты о выполненной работе;

1 Аналитический раздел

Перед теоритическим изложением алгоритмов, представленных в работе, требуется ввести понятия *редукционного расстояния* и *метода динамического программирования.*

Редукционное расстояние(расстояние Эйнштейна) – это минимальное количество редукционных операций, необходимых для преобразования одной строки в другую.

Есть следующие редукционные операции:

— Операции, вес которых - 1:
  I - insert(вставка);
  D - delete(удаление);
  R - replace(замена);
— Операция, вес которой - 0:
  M - match(совпадение);

Так минимальное расстояние между строками minD('увлечение', 'развлечение') = 3, но чтобы найти это, требуется перебрать расстояния с разным выравниваем строк по отношеню друг к другу.

		у	в	л	е	ч	е	н	и	е
р	а	з	в	л	е	ч	е	н	и	е
I	I	R	M	M	M	M	M	M	M	M

Проблема выравнивания решается рекуррентно через расстояния между подстроками фиксированной длины.

1.1 Описание Алгоритмов

Первым появившимся алгоритмом был алгоритм Левенштейна, который заложил фундамент в поиске расстояния между строками.

1.1.1 Расстояние Левенштейна

Расстояние Левенштейна имеет широкую область применения. Алгоритм используется в:

— поисковых системях, в базах данных, в автоматическом распознавание текста и речи для исправления ошибок и опечаток в слове;
— в утилитах для сравнения файлов(таких как diff);
— в биоинформатике для сравнения генов, хромосом и белков;

Алгоритм рекуррентно через расстояния между подстроками i и j находит расстояние между строками s1 и s2. Проверяя, какое действие будет наиболее выгодным I(insert), D(delete), M(match) или R(replace):

$$D(s1, s2) = D(s1[1..i], s2[1..j]) = min\begin{pmatrix} \text{D(s1[1..i], s2[1..j-1]) + 1,} \\ \text{D(s1[1..i-1], s2[1..j]) + 1,} \\ \text{D(s1[1..i-1], s2[1..j-1]) + } \left[\begin{array}{c} \text{0, если s1[i] = s2[j],} \\ \text{1, иначе} \end{array} \right. \end{pmatrix} \tag{1.1}$$

,где i – длина подстроки строки s1, которая изначально равно длинне s1, j – длина подстроки строки s2, которая изначально равно длинне s2

Таким образом, применив метод динамического програмированая мы разбиваем нашу задачу на небольшие подзадачи, которые достаточно легко можно решить. Данный метод удобно представлять в виде матрицы.

	λ	М	Г
λ	0	1	2
М	0	0	1

[1]

В виде матрицы редукционные операции можено представить следующим образом:

— I(insert) $\rightarrow$;
— D(delete) $\downarrow$;
— M(match) or R(replace) $\searrow$;

У данного алгоритма есть улучшенная версия, которую модифицировал Фредерик Дамерау.

1.1.2 Расстояние Дамерау – Левенштейна

Данный алгоритм применятеся также как и обычный в:

— поисковых системах;
— биоинформатике(сравнение белков линейной структуры);

Причиной появления данного алгоритма было огромное количество ошибок ввода – ввод двух соседних символов не в том порядке. Отсюда появляется новая операция в дополнении к уже имеющимся:

— X - exchange(или T - transposition);

[1](Представление D('М', 'МГ'))

Отсюда, формула 1.1 переходит в:

$$D(s1[1..i], s2[1..j]) = min \left(\begin{array}{c} \text{D(s1[1..i], s2[1..j-1])} + 1, \\ \text{D(s1[1..i-1], s2[1..j])} + 1, \\ \text{D(s1[1..i-1], s2[1..j-1])} + \left[\begin{array}{c} \text{0, если s1[i]} = \text{s2[j],} \\ \text{1, иначе} \end{array} \right. , \\ \text{D(s1[1..i-2], s2[1..j-2])} + 1 \end{array} \right) \tag{1.2}$$

Причем последняя операция(X) выполняется, если такая перестановка необхадима и требуется, и это не совпадение.

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.

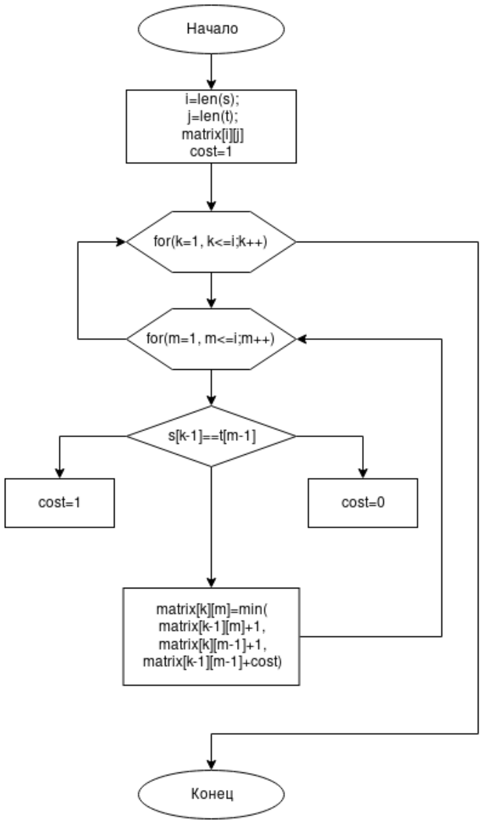

Рисунок 2.1 — Представлена схема алгоритма нахождения расстояния Левенштейна для матричной реализации

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.

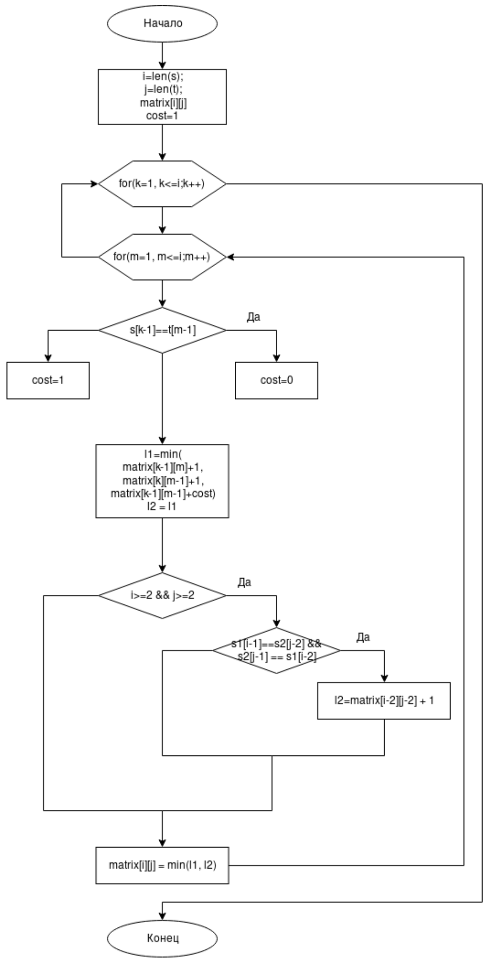

Рисунок 2.2 — Представлена схема алгоритма нахождения расстояния Дамерау – Левенштейна для матричной реализации

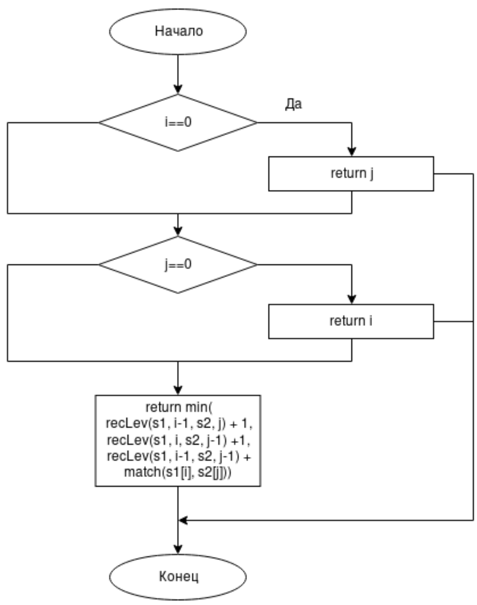

Рисунок 2.3 — Представлена схема алгоритма нахождения расстояния Левенштайна для рекурсивной реализации

3 Технологический раздел

Данный раздел содержит информацию о реализации ПО и листингах програм.

3.1 Требования к програмному 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:

```
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 алгоритма Дамерау – Левенштейна:

```
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 алгоримта Левенштейна в рекурсивной реализации:

```
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 функции match():

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

4 Экспериментальный раздел

Данный раздел посвящен тестированию и исследованию трех реализованных алгоримтов: 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
```

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

Алгоритм Левенштейна(Рекурсивный):

```
Input s1: MGTU
Input s2: MTGU
Length is : 2
```

```
Input s1: OTAPA
Input s2: TAPTAP
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, бессмысленно проводить эксперемент с длинной строк от 100 до 1000 на этом алгоритме. По-этому 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 несколько раз). Если обратить внимание на Левенштейна и Дамерау – Левенштейна, можно

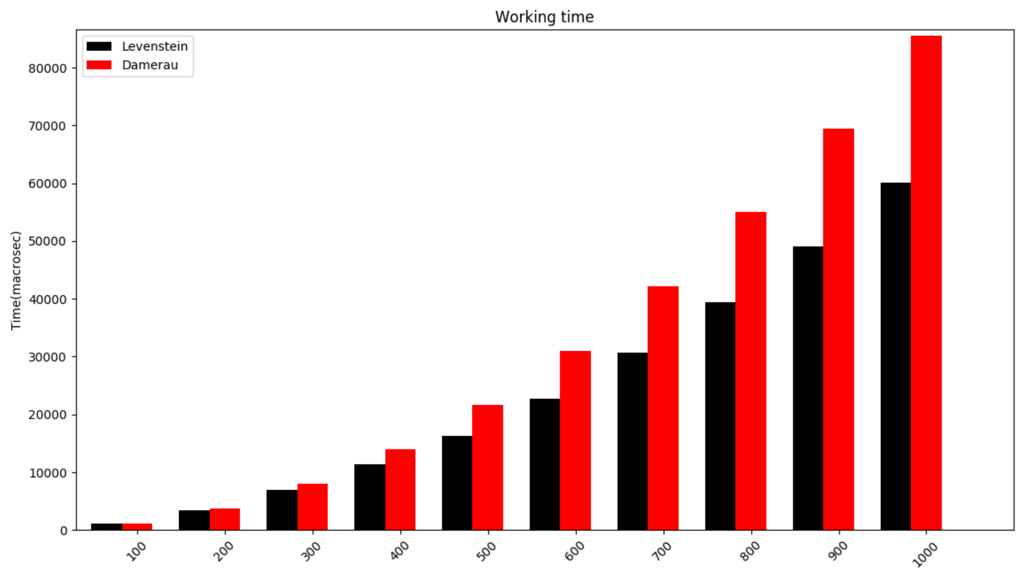

Рисунок 4.1 — Эксперемент первый, демонстрация различия времени для алгоритмов Левенштейна и Дамерау – Левенштейна

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.

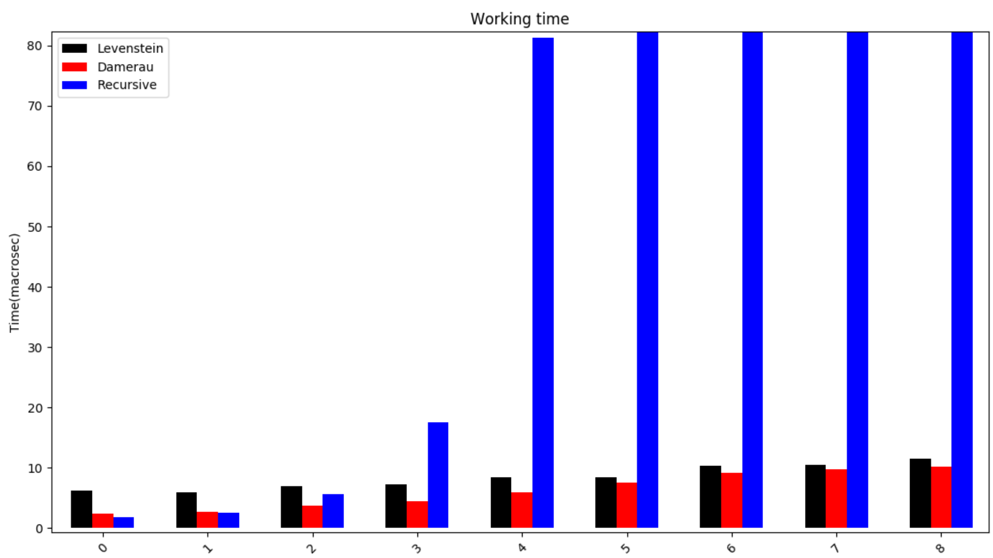

Рисунок 4.2 — Эксперемент второй, демонстрация различия времени для алгоритмов Левенштейна и Дамерау – Левенштейна и рекурсивной реализации Левенштейна

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
```

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):

```
==21489== HEAP SUMMARY:
==21489==     in use at exit: 0 bytes in 0 blocks
==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