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):

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

,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:

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

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.

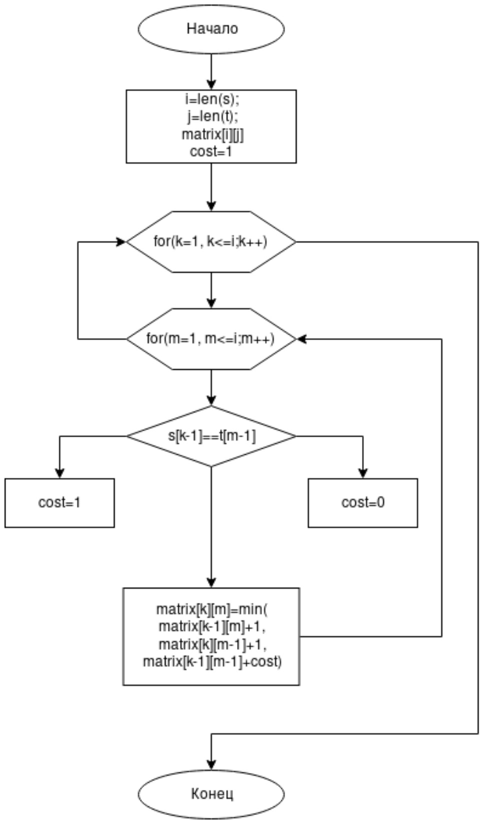

Рисунок 2.1 — Представлена схема алгоритма нахождения расстояния Левенштейна для матричной реализации

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.

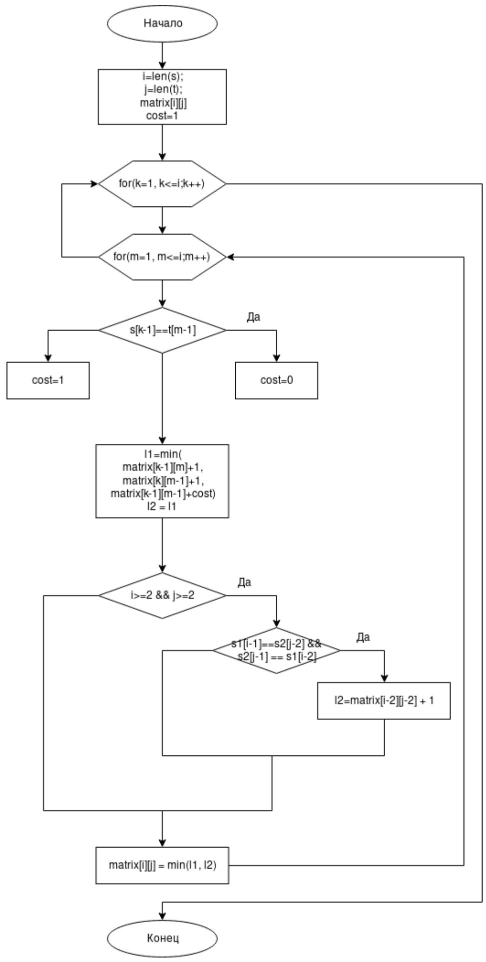

Рисунок 2.2 — Представлена схема алгоритма нахождения расстояния Дамерау – Левенштейна для матричной реализации

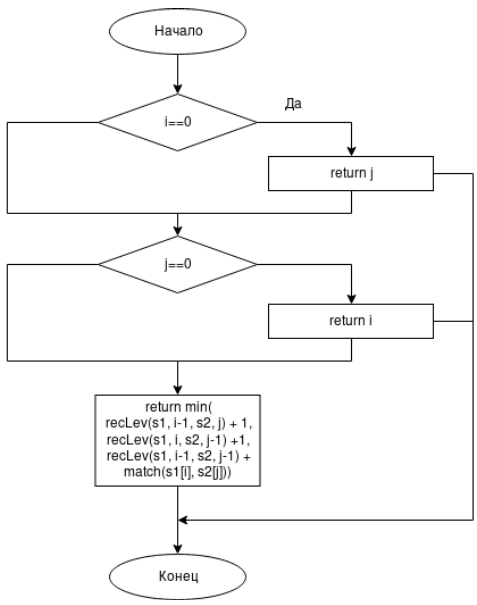

Рисунок 2.3 — Представлена схема алгоритма нахождения расстояния Левенштайна для рекурсивной реализации

3 Технологический раздел

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.

3.1 Листинг

Алгоритм Левенштейна:

```c
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:

```c
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 Исследовательский 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