МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ПРОЦЕССОВ УПРАВЛЕНИЯ

ЛАБОРАТОРНАЯ РАБОТА

По курсу:

«Алгоритмы и анализ сложности»

На тему:

«Эмпирический анализ алгоритма вычисления расстояния Дамерау-Левенштейна»

Работу выполнил студент
3 курса очного отделения:
Докиенко Денис Александрович

Санкт-Петербург
2019

Структура работы:

1. Введение

Зачастую вычислительные машины рассматриваются как средства для выполнения различных операций с числами и, с точки зрения их внутреннего устройства это совершенно логично. Но в жизни люди используют не только числа, но и различные символами, в том числе буквы или иероглифы естественных языков. Поэтому возникла необходимость некоторым образом представлять символы в вычислительных машинах.

В современной реализации вычислительных машин символы представляются в виде некоторого числового кода и в ряде задач это достаточно удобно. К примеру, для сравнения двух строк равной длины достаточно сравнить коды символов, располагающихся на одинаковых позициях в этих словах.

Но при этом существуют задачи, для решения которых представление символов в виде кодов не так удобно. Одной из таких задач является задача вычисления расстояния, то есть некоторой меры разности, между последовательностями символов (словами). Необходимость в вычислении расстояния между словами возникает при поиске ошибок в тексте, при классификации или кластеризации текстов, а также ещё во множестве прикладных задач.

При этом очевидно, что для вычисления расстояния между двумя словами неправильно будет просто посчитать сумму разностей кодов их соответствующих символов или использовать иные алгоритмы, основанные на

вычислении разности кодов символов. Поэтому необходимо вводить функцию разности двух слов, определённую иным образом. Одной из таких функций является **расстояние Дамерау-Левенштейна**.

2. Постановка задачи

Таким образом, расстояние Дамерау-Левенштейна — это мера разности двух конечных символьных строк.

Оно вычисляется как *минимальное количество* элементарных операций

- *Вставки* (добавления в строку на любую позицию одного символа)
- *Удаления* (удаления из строки одного любого символа)
- *Замены* (замены одного любого символа строки любым другим символом)
- *Транспозиции* (перестановки любых двух соседних символов строки)

необходимых для перевода одной строки в другую.

Изначально В. И. Левенштейн исследовал последовательности, состоящие из нулей и единиц в статье 1966 года «Двоичные коды с исправлением выпадений, вставок и замещений символов», рассматривая только операции вставки, удаления и замены. Позже более общую постановку задачи с последовательностями любых символов связали с его именем.

До этого в 1964 году Фредерик Дамерау опубликовал статью «A technique for computer detection and correction of spelling errors», в которой он привел исследование, показывающее, что более 80% человеческих ошибок (опечаток), попадает в один из четырёх классов (одна буква отсутствует, одна буква лишняя, одна буква заменена на другую, две соседних буквы переставлены местами).

Расстояние Дамерау-Левенштейна является логичной модификацией расстояния Левенштейна, основанной на исследовании Ф. Дамерау. То есть, к элементарным операциям вставки, удаления и замены в нём добавляется операция транспозиции.

Также нельзя не отметить, что расстояние Дамерау-Левенштейна часто примененяется в биоинформатике. И отчасти из-за этого возникла необходимость в присвоении элементарным операциям разных весов, что отражает различную вероятность мутации в биологии или, если возвращаться к компьютерным наукам - различную вероятность опечаток в тексте.

При этом считается, что веса должны быть неотрицательны и для них действует правило треугольника: если несколько последовательных операций возможно заменить на одну операцию, то это не ухудшает стоимость рассматриваемых операций.

Для простоты выкладок и учитывая акцент приведённой работы на анализе сложности алгоритма, а не на подробном исследовании его различных модификаций, будем считать веса всех операций равными единице. При этом все рассуждения описанные здесь можно обобщить на случай использования весов различной стоимости.

3. Обзор алгоритма

Итак, задачу поиска расстояния Дамерау-Левенштейна можно определить как задачу минимизации **[1]**:

$$1 * dCount + 1 * iCount + 1 * rCount + 1 * tCount \rightarrow min$$

Где $dCount, iCount, rCount, tCount$ - это количество элементарных операций ($d - delete, i - insert, r - replace, t - transpose$), а вместо единиц могут быть указаны иные веса операций.

Эта задача решается с помощью модифицированного алгоритма Вагнера-Фишера. Оригинальный алгоритм был предложен Р. Вагнером и М. Фишером в статье «The string-to-string correction problem» 1974 года и предназначается для вычисления расстояния Левенштейна (то есть расстояния Дамерау-Левенштейна без элементарной операции транспозиции: $1 * tCount$). Соответственно, для вычисления расстояния Дамерау-Левенштейна алгоритм модифицируется таким образом, чтобы он учитывал также и возможность использования элементарной операции транспозиции.

Приведем неформальное описание оригинального алгоритма Вагнера-Фишера, а в следующих параграфах обсудим его более подробно и произведём необходимые для вычисления расстояния Дамерау-Левенштейна модификации:

Задача поиска расстояния Левенштейна состоит в нахождении последовательности элементарных операций вставки, удаления и замены, производимых над одной строкой и преобразующей её ко второй строке. При этом данная последовательность операций должна удовлетворять условию оптимальности: сумма **[1]** должна быть на ней минимальной.

Для решения этой задачи можно использовать динамическое программирование:

Для определённости допустим, что мы преобразуем строку A к строке B, то есть изменяем только строку A. Ясно, что это не обязательное условие при поиске оптимального решения, то есть можно преобразовывать строку B вместо строки A, либо преобразовывать обе строки, сводя их к некоторой третьей строке. При этом ни один из этих вариантов не предпочтителен с точки зрения оптимальности полученного решения.

В дальнейшем строку A будем называть **изменяемой**, а строку B - **целевой**.

Рассмотрим последние символы строк и допустим, что они не совпадают между собой (случай совпадения последних символов рассматривается ниже). Тогда, в соответствии с определением расстояния Левенштейна, для того, чтобы преобразовать строку A к строке B, можно либо удалить последний символ строки A, либо заменить его символом, стоящим на конце строки B, либо вставить последний символ строки B в конец строки A. **[2]**

Тогда возникает три различных класса решений задачи. Каждое решение в этих классах состоит из некоторой последовательности элементарных операций, которая заканчивается одной из трёх указанных операций соответственно своему классу.

Предположим, что мы можем выбрать по одному решению из этих трёх классов так, чтобы последовательность операций, предшествующая рассматриваемой последней операции, была оптимальной. Тогда оптимальное из таких трёх решений и будет оптимальным для всей задачи.

Хорошо прослеживается рекурсивная природа данного алгоритма. Для нахождения трёх решений, из которых мы затем отбираем конечное, можно использовать тот же алгоритм, указав в качестве входных данных префиксы изначальных строк. Таким образом длина префиксов будет постепенно уменьшаться и задача сойдётся к одному из трёх тривиальных случаев, оптимальная последовательность операций для которых очевидна **[3]**:

- Длина обеих строк равна нулю. В этом случае никаких действий для преобразования одной строки к другой совершать не нужно.

- Длина преобразуемой строки равна нулю, при этом длина целевой - ненулевая. В этом случае для преобразования строки необходимо в неё вставить все символы из целевой строки. То есть оптимальная последовательность операций - это N операций вставки, где N - длина целевой строки.

- Длина преобразуемой строки не равна нулю, а длина целевой - нулевая. В этом случае необходимо удалить все символы из преобразуемой строки.

То есть оптимальная последовательность операций - это M операций удаления символа, где M - длина преобразуемой строки.

Из описания алгоритма видно, что в случае совпадения последних символов в рассматриваемых строках, можно сразу переходить к решению задачи на префиксах с меньшими на единицу размерами. **[4]**

Таким образом, решение задачи сводится к последовательному решению более простых задач, начальные из которых (для трёх тривиальных случаев) имеют очевидное решение.

В программной реализации алгоритма значения функции расстояния можно записать в матрицу, индексы которой будут соответствовать длинам рассматриваемых префиксов, а в качестве значений будут записаны расстояния между ними. Назовём эту матрицу **матрицей решения**.

При этом в матрице решения необходимо записать также три тривиальных случая, то есть случаи, когда хотя бы одна из строк пустая. Учитывая соответствие индексов матрицы длине префиксов, значения на тривиальных случаях будут содержаться в первых строке и столбце матрицы (с индексом [0]).

Тогда для решения поставленной задачи необходимо вычислять значения матрицы решения по строкам или столбцам, переходя от меньших префиксоов к большим. Последний элемент матрицы (правый нижний элемент) будет содержать конечный ответ.

4. Формальное описание

Воспользуемся обозначениями введёнными выше. Также обозначим за $DL(A,B)$ функцию, принимающую на вход две строки и возвращающую расстояние Левенштейна между ними. Тогда можно сказать, что $DL(A,B)=d(|A|,|B|)$, где $|A|,|B|$ - длины строк, а $d(i,j)$ - расстояние между префиксами строк A и B, то есть между первыми i символами строки A и первыми j символами строки B.

Будем нумеровать элементы элементы строк с единицы, а не с нуля. При этом в матрице решения нулевой индекс будет соответствовать пустой строке. То есть, к примеру, элемент с индексом [0][X] будет содержать расстояние между пустой строкой и соответствующим префиксом длиной X.

Функция $d(i,j)$ задаётся рекурсивно следующим образом **[5]**:

- $0, if\ i=0, j=0$. Расстояние между двумя пустыми строками очевидно равно нулю.

- $i, if\ i>0, j=0$. Для того, чтобы получить из строки длиной i пустую строку, необходимо произвести i операций удаления.

- $j, if\ i=0, j>0$. Для того, чтобы получить из пустой строки строку длиной j , необходимо произвести j операций вставки.

- $d(i-1, j-1), if\ A[i]=B[j], i>0, j>0$. Если последние символы префиксов равны, то никаких операций с ними совершать не нужно, поэтому можно сразу переходить к вычислению функции от префиксов с удалёнными последними элементами .

- $min(d(i, j-1)+1; d(i-1, j)+1; d(i-1, j-1)+1), if\ A[i]\neq B[j], i>0, j>0$ **[6]**. Если последние символы префиксов не равны, то возможно:

  a) Вставить в конец преобразуемой строки тот же символ, что стоит на конце целевой.

  b) Удалить последний символ целевой строки.

  c) Заменить последний символ преобразуемой строки последним символом целевой

Описание данной формулы приведено в предыдущем параграфе, поэтому лишь кратко опишем её соответствие со словесными пояснениями.

Первые три случая соответствуют тривиальным случаям рассчёта расстояния между словами, когда хотя бы одно из слов пустое **[3]**. Четвёртый случай описывает ситуацию, когда последние символы рассматриваемых префиксов совпадают. Тогда можно перейти сразу к оценке расстояния между префиксами меньшего на единицу размера **[4]**. Пятый случай описывет три различные элементарные операции, которые возможно совершить на текущем шаге работы алгоритма, чтобы сравнять исходные строки **[2]**.

Приводить доказательство корректности данной формулы здесь, вероятно, будет излишним, но при необходимости с ним можно ознакомиться в оригинальной статье.

Также приведём псевдокод алгоритма:

```
D[0][0] = 0; # Нулевой элемент матрицы решения

# N,M – длины строк S1 (преобразуемой) и S2 (целевой)

for i = 1 to N do {

    D[i][0] = i; }
```

```
for j = 1 to M do {
    D[0][j] = j; }
for i = 1 to N do {
    for j = 1 to M do {
        if S1[i] != S2[j] then {
            m1 = D[i][j-1] + 1;
            m2 = D[i-1][j] + 1;
            m3 = D[i-1][j-1] + 1;
            D[i][j] = min(m1,m2,m3); }
        else D[i][j] = D[i-1][j-1];
    }
}
return D[N][M];
```

Прим.: в оригинальной статье данный алгоритм приведён под названием *Algorithm X* и не рассматривает случай совпадения последних символов префиксов.

Приведём небольшое пояснение к данному коду:

Сначала алгоритм заполняет элементы матрицы решения, соответствующие тривиальным случаям (элемент с индексом [0][0] и нулевые строку и стобец матрицы).

Затем последовательно по строкам (но можно вычислять и по столбцам) вычисляются остальные элементы матрицы. Последним заполняется элемент матрицы с индексом [N][M] (N,M - длины строк) - вычисляемое расстояние Левенштейна.

5. Модификация

Итак, теперь необходимо модифицировать алгоритм так, чтобы учитывать также элементарную операцию транспозиции:

Рассмотрим две строки $C[0],C[1],\dots,C[N]$ и $T[0],T[1],\dots,T[M]$ (C - converted, T - target). Пусть в строку C входит символ $T(M)$, а в строку T - $C(N)$.

Мы можем найти индексы их последнего вхождения в строки. Пусть это будет n и m соответственно. Тогда для того, чтобы поменять местами $T(M)$ и $C(N)$ в строке C нужно:

- Удалить все символы между $C(N)$ и $C(n)$
- Поменять местами $C(N)$ и $C(n)$, ставшие соседними
- Вставить в строку C все символы, лежащие между $T(M)$ и $T(m)$

Очевидно, что после выполнения указанных действий строки C и T будут совпадать до префиксов длиной $n-1$ и $m-1$.

Иными словами, после выполнения операции транспозиции и сопутствующих операций удаления и вставки для некоторых префиксов i и j можно перейти к вычислению функции для префиксов длины n и m.

Тогда конечная формула вычисления стоимости операции транспозиции следующая:

$$d(n,m)+(i-n-1)+1+(j-m-1)$$

Следовательно, для учёта операции транспозиции можно рассчитать $d(i,j)$ по формуле **[5]** и по данной формуле, после чего выбрав наименьший по стоимости вариант.

Реализация этой модификации достаточно тривиальна, поэтому приводить псевдокод модифицированного алгоритма не будем. Единственное, что требует отдельного описания — это нахождение индексов n и m последних вхождений символов в строки.

Очевидно, что малоэффективно на каждой итерации алгоритма производить поиск последних вхождений рассматриваемых символов, поэтому будем хранить их заранее.

Для хранения индекса n необходимо завести некоторую структуру данных, которая указывает последнее вхождение каждого символа алфавита (множества всех используемых в строках символов) в преобразуемую строку. Она изначально заполняется нулям, а затем в конце каждой итерации внешнего цикла (то есть при проходе каждой строки матрицы решения в приведённом выше псевдокоде) обновляет значение для рассматриваемого символа преобразуемой строки.

Индекс m указывает на последнее вхождение рассматриваемого в рамках внешнего цикла символа (рассматриваемого последнего символа преобразуемой

строки), то есть его можно хранить в единственной переменной, перезаписывая только в случае совпадения последних символов строк и обнуляя в начале каждого внешнего цикла.

6. Математический анализ алгоритма

Итак, представленный алгоритм реализован с использованием двух вложенных циклов размерностью N и M, которые задают размеры строк, между которыми и вычисляется расстояние Дамерау-Левенштейна.

При этом алгоритм заключается в последовательном вычислении элементов матрицы решений, имеющей размерность $N \times M$. Из этого описания можно определить сложность алгоритма: $O(N * M)$

Из формулы **[5]** видно, что в случае совпадения на шаге алгоритма последних элементов рассматриваемых префиксов каких-либо вычислений не производится (значение равно одному из предыдущих элементов матрицы). Тогда очевидно, что в случае полного совпадения рассматриваемых строк алгоритм будет работать быстрее всего.

Если последние элементы строк различаются, то производится вычисление стоимости преобразования с помощью трёх элементарных операций (см. **[6]** в формуле **[5]**). Из этого можно было бы сделать вывод о том, что алгоритм будет работать медленнее всего при отсутствии совпадающих символов в обеих строках (то есть если обе строки состоят из различных символов). Но при этом важно также то, что в этом случае ни разу не будет вычислена стоимость операции транспозиции. В связи с этим нельзя однозначно определить, при каких входных данных будет наблюдаться наихудшая скорость работы приведённого алгоритма, а подробное исследование этого вопроса, вероятно, выходит за рамки данной работы.

7. Описание эксперимента и входных данных

Для проверки правильности вышеизложенных рассуждений проведём следующий эксперимент:

На вход алгоритму будем подавать слова одинаковых длин, замеряя время исполнения программы в секундах. Т. к. в этом случае N и M будут равны, то есть предполагаемая сложность алгоритма - $O(N^2)$, то ожидаемая зависимость времени исполнения программы от размера входных слов - квадратичная.

Также будем подавать на вход алгоритму слова разных длин для проверки предположения об одинаковой зависимости сложности алгоритма от длин подаваемых на вход слов.

Слова генерируются случайным образом и состоят только из строчных букв английского алфавита.

Очевидно, что время работы алгоритма зависит в том числе и от используемого для генерации слов алфавита (набора символов). Тем не менее, в рамках исследования именно класса сложности алгоритма это не имеет значения, т. к. сам вид рассматриваемой зависимости от алфавита очевидно не зависит.

В ходе эмпирической оценки времени работы алгоритма для проведения эксперимента были выбраны следующие параметры входных данных:

- Шаг - 10
- Диапазон входных данных:
    1. [1, 201] в случае равенства длин входных слов
    2. [1, 101] в случае разных длин входных слов

Также произведём аналогичный эксперимент для входных слов равной длины в случае удвоения размера входных данных. Тогда параметры входных данных будут следующие:

- Шаг - 20
- Диапазон входных данных - [2, 402]

8. Генерация входных данных и реализация алгоритма

Алгоритм и генератор входных данных доступны по следующей ссылке на Github: https://github.com/sncodeGit/Other/blob/master/DamLevDist.py.

Функция getDamLevDist(C, T) принимает на вход две строки и возвращает расстояние Дамерау-Левенштейна между ними. Для вычисления времени работы в её код добавлено четыре строки с рассчётом значения глобальной переменной runtime.

Функция getStrings(N, M, seed) принимает на вход размеры генерируемых слов и seed (параметр генерации псевдослучайных чисел), а возвращает две строки, состоящие только из строчных букв английского алфавита. При этом каждый символ располагается на любой позиции в слове равновероятно.

В функции main() реализован сам вычислительный эксперимент, результаты которого описаны в следующем параграфе. В её коде вызывается функция getRuntime(C, T, iteration_num), которая по строкам C и T iteration_num раз

вычисляет расстояние Дамерау-Левенштейна и возвращает среднее время работы алгоритма.

9. Результаты вычислительного эксперимента

Результаты измерения времени работы алгоритма на различных длинах обоих входных слов (то есть при словах равных длин) приведены в следующей таблице. Также в ней указано отношение времени работы алгоритма на удвоенных входных данных (из диапазона [2, 402]) к времени работы на данных из диапазона [1, 201]:

Табл.: Результаты эксперимента при словах равных длин (N — длины слов, Runtime - время работы алгоритма)

N1	Runtime1 (сек.)	N2	Runtime2 (сек.)	Runtime2 / Runtime1
1	0.00003552436828613	2	0.0001553879547119	4.37
11	0.00163233280181884	22	0.0062334060668945	3.82
21	0.00707995891571044	42	0.0301726126861572	4.26
31	0.01215059757232666	62	0.0512350196838378	4.22
41	0.02363080978393554	82	0.09041216850280	3.83
51	0.03921124935150146	102	0.15267765998840	3.89
61	0.05498099327087402	122	0.2374683570861	4.32
71	0.0802231788635254	142	0.33529186248723	4.18
81	0.1059417724609375	162	0.4293544101715	4.05
91	0.1323167324066162	182	0.5467164230346	4.13
101	0.16235713958740233	202	0.76504821777343	4.71
111	0.20442919731140136	222	0.7831439971923	3.83
121	0.24882566928863525	242	1.0081685066223	4.05
131	0.2934983491897583	262	1.354333686828613	4.61
141	0.3453718423843384	282	1.358984642028808	3.93
151	0.3977543592453003	302	1.63411822319030	4.11
161	0.4536875247955322	322	1.800924869537353	3.97
171	0.509886360168457	342	2.03616490364074	3.99
181	0.5859748840332031	362	2.577913904190063	4.4
191	0.6568373918533326	382	2.20651087760925	3.36
201	0.7231623888015747	402	3.387547397613525	4.68

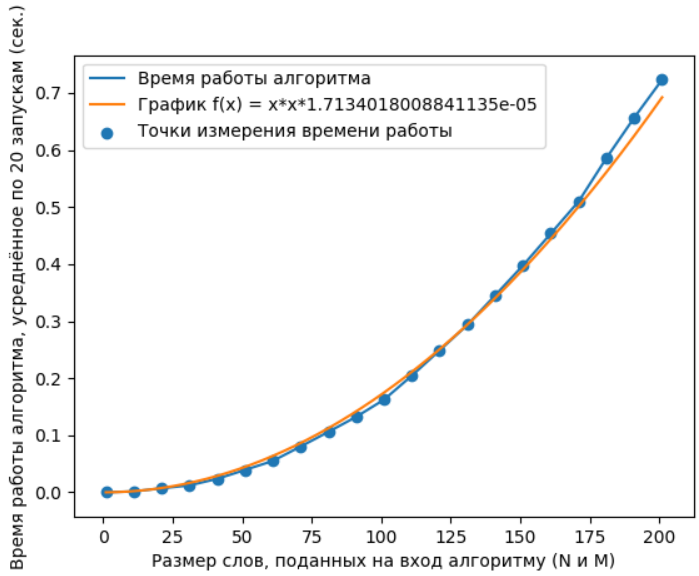
Каждое время работы алгоритма при длинах входных слов, лежащих в отрезке [1,201], усреднялось на 20 запусках алгоритма. Время работы при длинах слов из отрезка [2, 402] усреднялось на 10 запусках алгоритма (т. к. вычисления происходят достаточно долго).

Также привожу графики, построенные по данным точкам. На рисунке они аппроксимированы параболами с коэффициентами, найденным как среднее арифметическое частных от времени работы алгоритма и квадрата размера входных слов (коэффициенты указаны на графиках):

$$\frac{\sum_i N[i]/Runtime[i]}{21}$$

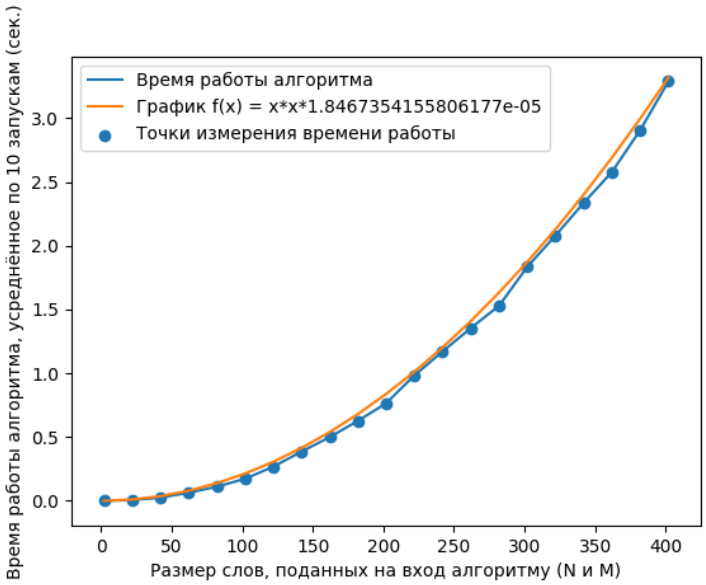
Первый график построен на основании данных из столбцов N1 и Runtime1:

Рис. 1: График зависимости времени работы алгоритма от длины обоих слов



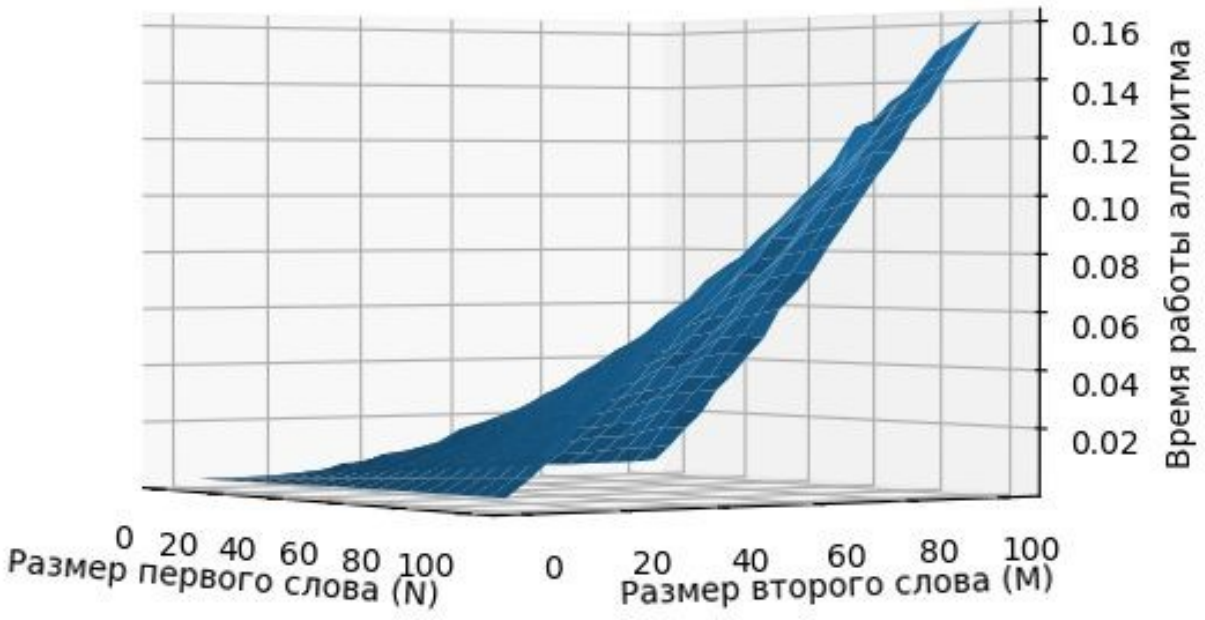
Второй график построен на основании столбцов таблицы N2 и Runtime2:

Рис. 2: График зависимости времени работы алгоритма от длины обоих входных слов при удвоенных входных данных



Также был построен трёхмерный график зависимости времени работы алгоритма от размеров входных слов:

Рис. 3: График зависимости времени работы алгоритма от длины входных слов



Приведённые графики построены с помощью библиотеки matplotlib и при необходимости их можно построить самостоятельно с помощью кода, указанного в функции main().

10. Анализ полученных данных

Из графиков видно, что скорость работы алгоритма действительно можно описать квадратичной зависимостью от размеров входных слов.

Также из таблицы можно высчитать среднее арифметическое отношения времени работы алгоритма на удвоенных входных данных и на обычных:

$$\frac{\sum_i Runtime\,2[i] / Runtime\,1[i]}{21}$$

В данном случае это значение приблизительно равняется 4.13. При этом, в случае квадратичной зависимости сложности алгоритма от размера входных данных, при их удвоении время работы алгоритма должно увеличиваться в четыре раза.

Приведённые выше результаты подтверждают выводы, сделанные в шестом параграфе.

Также видно, что из-за усреднения результатов скорости работы алгоритма на двадцати и десяти запусках соответственно была получена достаточно хорошая точность аппроксимации дискретных результатов измерения параболой.

11. Характеристики вычислительной среды

ОС: Linux с версией ядра: 4.15.0-72 (Ubuntu 16.04.1)

Процессор: AMD Radeon R4 Graphics.

Размер ОЗУ: 4 Гб.

IDE: Spyder3

12. Список используемой литературы

- В. И. Левенштейн, “Двоичные коды с исправлением выпадений, вставок и замещений символов”, *Докл. АН СССР*, 163:4 (1965), 845–848
- The string-to-string correction problem. RA Wagner, MJ Fischer. Journal of the ACM (JACM) 21 (1), 168-173, 1974
- Damerau, F.J.: A technique for computer detection and correction of spelling errors. Commun. ACM 7(3), 171–176 (1964)
- Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ — 3-е изд. — М.: «Вильямс», 2013. — с. 440. — ISBN 978-5-8459-1794-2
- Navarro, Гонсало (март 2001 г.), "Экскурсия к приближенному совпадения строк", ACM Computing Surveys
- https://spravochnick.ru/informatika/metod_levenshteyna/