

Государственное образовательное учреждение высшего профессионального образования
“Московский государственный технический университет имени Н.Э.Баумана”

Дисциплина: АНАЛИЗ АЛГОРИТМОВ

СЕМИНАР - КОНВЕЙЕРНЫЕ ВЫЧИСЛЕНИЯ

ТЕМА-расстояния Левенштейна и Дамерау-Левенштейна.

Студент группы ИУ7-55,
Шестовских Николай Александрович

2019 г.

Содержание

Введение	2
1 Аналитическая часть	3
1.1 Описание алгоритмов	3
1.2 Вывод	3
1.3 Вывод	4
2 Технологическая часть	5
2.1 Требования к программному обеспечению	5
2.2 Средства реализации	5
2.3 Листинг кода	5
2.4 Тестирование	7
2.5 Сравнительный анализ потребляемой памяти	7
2.6 Оценка потребляемой памяти на 4 и 1000 символах	8
2.7 Вывод	10
3 Экспериментальная часть	11
3.1 Постановка эксперимента	11
3.2 Результаты эксперимента	11
3.3 Вывод	12
Заключение	13
Список литературы	14

Введение

Конвейер придумали в 20 веке. Лабораторная работа на передачу данных между потоками.
 t_i

1 Аналитическая часть

1.1 Описание алгоритмов

Обозначим простейшие действия над двумя строками:

1. D - удалить букву в одной из строк;
2. I - вставить букву в одной из строк;
3. R - заменить букву в одной из строк;
4. M - совпадение двух букв.

Каждая операция имеет свою ‘цену’ - у совпадения она 0, а у трех остальных - по 1. Задача нахождения расстояния Левенштейна между двумя строками сводится к поиску набора операций, которые нужно совершить, чтобы трансформировать одну строку в другую, причем их суммарная цена должна быть минимальной.

Расстояние Дамерау-Левенштейна отличается от расстояния Левенштейна добавлением операции транспозиции - перестановки двух соседних символов (ее цена - 1).

Пусть S_1 и S_2 — две строки над некоторым алфавитом, тогда расстояние Левенштейна между ними можно подсчитать по следующей формуле (1):

$$D(i, j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min(D1, D2, D3) & \end{cases} \quad (1)$$

где:

$D1 = D(i, j-1) + 1$;

$D2 = D(i-1, j) + 1$;

$D3 = D(i-1, j-1) + (0 \text{ если } \text{match}(S_1[i], S_2[j]), 1 \text{ иначе})$;

$\text{match}(a, b)$ = истина при $a = b$, ложь иначе;

$\min(A1, A2, \dots, AN)$ - минимум среди чисел $A1, A2, \dots, AN$.

В свою очередь, расстояние Дамерау-Левенштейна можно посчитать по формуле (2):

$$D(i, j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min(D1, D2, D3, D4), & i > 1, j > 1, a_i = b_{j-1}, a_{i-1} = b_j \\ \min(D1, D2, D3) & \end{cases} \quad (2)$$

где:

$D4 = D(i-2, j-2) + 1$, если $i > 1, j > 1$ и $a_i = b_{j-1}, a_{i-1} = b_j$.

1.2 Вывод

В данном разделе были рассмотрены формулы для нахождения расстояния Левенштейна и Дамерау-Левенштейна, а также основные применения этих расстояний.

1.3 Вывод

В данном разделе были рассмотрены схемы алгоритмов нахождения расстояния Левенштейна, Дамерау-Левенштейна, а также рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна.

2 Технологическая часть

2.1 Требования к программному обеспечению

Входные данные: str1 - первое слово, str2 - второе слово.

Выходные данные: значение расстояния между двух слов.

2.2 Средства реализации

В качестве языка был выбран python, для вычисления памяти был использован метод `sys.getsizeof()`, возвращающий память, затрачиваемую на объект. Для замеров времени была использована функция `tick()`, приведенная ниже:

Листинг 1: Функция замера времени

```
1 unsigned long long tick(void)
2 {
3     unsigned long long d;
4     __asm__ __volatile__ ("rdtsc" : "=A" (d) );
5     return d;
6 }
```

Для подключения ее в модуль python была создана библиотека `libtick.so` и подключена с помощью модуля `ctypes`.

2.3 Листинг кода

Листинг 2: Функция нахождения расстояния Левенштейна итеративно

```
1 def Livenstein_matr(s1, s2):
2     l1 = len(s1)
3     l2 = len(s2)
4     matr = [[i + j for i in range(l2 + 1)] for j in range(l1 + 1)]
5
6     for i in range(1, l1 + 1):
7         for j in range(1, l2 + 1):
8             opt1 = matr[i][j - 1] + 1
9             opt2 = matr[i - 1][j] + 1
10            opt3 = matr[i - 1][j - 1] + (0 if s1[i - 1] == s2[j - 1] else 1)
11            matr[i][j] = min(opt1, opt2, opt3)
12
13     res = matr[l1][l2]
14     return res
```

Листинг 3: Функция нахождения расстояния Дамерау-Левенштейна итеративно

```

1  def Livenstein_Damerau_matr(s1, s2):
2      l1 = len(s1)
3      l2 = len(s2)
4      matr = [[i + j for i in range(l2 + 1)] for j in range(l1 + 1)]
5
6      for i in range(1, l1 + 1):
7          for j in range(1, l2 + 1):
8              opt4 = math.inf
9              opt1 = matr[i][j - 1] + 1
10             opt2 = matr[i - 1][j] + 1
11             opt3 = matr[i - 1][j - 1] + (0 if s1[i - 1] == s2[j - 1] else 1)
12             if i > 1 and j > 1 and s1[i - 2] == s2[j - 1] and s1[i - 1] == s2[j - 2]:
13                 opt4 = matr[i - 2][j - 2] + 1
14
15             matr[i][j] = min(opt1, opt2, opt3, opt4)
16
17     res = matr[l1][l2]
18     return res

```

Листинг 4: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```

1  def Livenstein_Damerau_recur(s1, s2):
2      if s1 == "" and s2 == "":
3          return 0
4      elif s1 != "" and s2 == "":
5          return len(s1)
6      elif s1 == "" and s2 != "":
7          return len(s2)
8
9      l1 = len(s1)
10     l2 = len(s2)
11
12     opt4 = math.inf
13     opt1 = Livenstein_Damerau_recur(s1, s2[:-1]) + 1
14     opt2 = Livenstein_Damerau_recur(s1[:l1 - 1], s2) + 1
15     opt3 = Livenstein_Damerau_recur(s1[:l1 - 1], s2[:-1]) + (0 if s1[l1 - 1] == s2[l2 - 1]
16         else 1)
17     if l1 > 1 and l2 > 1 and s1[l1 - 1] == s2[l2 - 2] and s1[l1 - 2] == s2[l2 - 1]:
18         opt4 = Livenstein_Damerau_recur(s1[:l1 - 2], s2[:l2 - 2]) + 1
19
20     res = min(opt1, opt2, opt3, opt4)
21     return res

```

2.4 Тестирование

Было организовано тестирование с помощью заранее подготовленных данных в виде пары строк и расстояния по Левенштейну и Дамерау-Левенштейну. Они приведены в таблице 1:

Таблица 1. Тестовые данные для алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

s1	s2	р. Левенштейна	р. Дамерау-Левенштейна
'ser'	'gey'	2	2
'ser'	'sre'	2	1
'ser'	'guy'	3	3
'ser'	'esg'	3	2
'sergeyser'	'sergey'	3	3
"	'sergey'	6	6
"	"	0	0
"	'a'	1	1
'a'	'b'	1	1
'serg'	'segr'	2	1

Все реальные результаты совпали с ожидаемыми из таблицы 1, что подтвердило правильность работы программы.

2.5 Сравнительный анализ потребляемой памяти

Найдем память, затрачиваемую на объекты одного вызова каждой из функций:

Таблица 2. Память, потребляемая структурами в алгоритме Левенштейна

Структура данных	Занимаемая память (байты)
Матрица	$40 + 8 * \text{len}(\text{str1}) + 1 + \text{len}(\text{str1}) + 1 * [40 + 8 * (\text{len}(\text{str2}) + 1)]$
2 строки	$2 * [49 + \text{len}(\text{str})]$
5 вспомогательных переменных(int)	140
2 счетчика (int)	56

Таблица 3. Память, потребляемая структурами в алгоритме Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
Матрица	$40 + 8 * \text{len}(\text{str1}) + 1 + \text{len}(\text{str1}) + 1 * [40 + 8 * (\text{len}(\text{str2}) + 1)]$
2 строки	$2 * [49 + \text{len}(\text{str})]$
6 вспомогательных переменных(int)	166
2 счетчика (int)	56

Таблица 4. Память, потребляемая структурами в рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна за один вызов

Структура данных	Занимаемая память (байты)
4 вспомогательных переменных (int)	112
2 строки	$98 + \text{len}(\text{str1}) + \text{len}(\text{str2})$

Максимальная глубина рекурсивного вызова функции - сумма длин двух слов.

Суммируя память, затрачиваемую структурами данных в итеративном поиске расстояния Дамерау-Левенштейна по таблице 3 и принимая $\text{len}(s1) = \text{len}(s2) = l$, получим:

$$DLIMem = 8l^2 + 59l + 411, \quad (3)$$

где DLIMem - суммарная память, затрачиваемая структурами данных в итеративном поиске расстояния Дамерау-Левенштейна.

Также найдем суммарную память, затрачиваемую рекурсивным поиском расстояния Дамерау-Левенштейна, принимая $\text{len}(\text{str1}) = \text{len}(\text{str2}) = l/2$ и максимальное количество одновременных вызовов - глубину вызовов(сумма длин двух строк).

$$DLRMem = 2l^2 + 420l, \quad (4)$$

где DLRMem - суммарная память, затрачиваемая структурами данных в рекурсивном поиске расстояния Дамерау-Левенштейна.

Найдем значения l , при которых $DLRMem = DLIMem$:

$$8l^2 + 59l + 411 = 2l^2 + 420l$$

Вычтем из обеих частей правую, получим:

$$6l^2 - 361l + 411 = 0 \quad (5)$$

Найдем корни уравнения (5):

$$\begin{aligned} l &= \frac{361}{12} \pm \frac{\sqrt{120457}}{12} \\ l1 &= \frac{361}{12} + \frac{\sqrt{120457}}{12} \\ l2 &= \frac{361}{12} - \frac{\sqrt{120457}}{12} \end{aligned}$$

Исходя из того, что:

$$\begin{aligned} DLIMem > DLRMem &\Leftrightarrow 6l^2 - 361l + 411 > 0 \\ DLIMem < DLRMem &\Leftrightarrow 6l^2 - 361l + 411 < 0 \end{aligned}$$

Получаем:

$$\begin{aligned} DLIMem > DLRMem &\Leftrightarrow l \in (-\infty, l1) \cup (l2, +\infty) \\ DLIMem < DLRMem &\Leftrightarrow l \in (l1, l2) \end{aligned}$$

Из того, что $l1 \approx 1.09$, а $l2 \approx 59.08$ делаем вывод, что при длине строк в отрезке от 0 до 1 и в интервале от 59 до бесконечности итеративная реализация поиска расстояния Дамерау-Левенштейна потребляет больше памяти, чем рекурсивная, а в отрезке от 2 до 59 рекурсивная реализация более затратна по памяти. На рисунке 5 приведена зависимость потребляемой памяти в различных реализациях поиска расстояния Дамерау-Левенштейна.

2.6 Оценка потребляемой памяти на 4 и 1000 символах

Оценим алгоритмы на словах длиной 4 символа:

Таблица 5. Память, потребляемая структурами в алгоритме нахождения расстояния Левенштейна

Структура данных	Занимаемая память (байты)
Матрица	480
2 строки	106
5 вспомогательных переменных(int)	140
2 счетчика (int)	56
Всего	780

Таблица 6. Память, потребляемая структурами в алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
Матрица	480
2 строки	106
6 вспомогательных переменных(int)	168
2 счетчика (int)	56
Всего	808

Таблица 7. Память, потребляемая структурами в рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
4 вспомогательных переменных (int)	$140 * 8(\text{максимальная глубина вызовов}) = 896$
2 строки	$106 * 4/2(\text{Усредненное значение}) = 212$
Всего	1108

Оценим алгоритмы на словах длиной 1000 символов:

Таблица 8. Память, потребляемая структурами в алгоритме нахождения расстояния Левенштейна

Структура данных	Занимаемая память (байты)
Матрица	8 064 096
2 строки	106
5 вспомогательных переменных(int)	140
2 счетчика (int)	56
Всего	8064398

Таблица 9. Память, потребляемая структурами в алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
Матрица	8 064 096
2 строки	106
6 вспомогательных переменных(int)	168
2 счетчика (int)	56
Всего	8064426

Таблица 10. Память, потребляемая структурами в рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
4 вспомогательных переменных (int)	$140 * 2000$ (максимальная глубина вызовов) = 280000
2 строки	$2098 * 1000 / 2$ (Усредненное значение) = 1049000
Всего	1329000

По таблицам 5,6,8 и 9 мы видим, что и при длине слов 4, и при длине слов 1000 итеративные реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна сравнимы по потребляемой памяти, причем разница между ними всегда 28 байт. В свою очередь, рекурсивная реализация Дамерау-Левенштейна при 4 потребляет больше памяти, а при 1000 - меньше, что видно в таблицах 6, 7, 9 и 10.

2.7 Вывод

В технологической части были предоставлены реализации алгоритмов нахождения расстояния Левенштейна в итеративной форме, а также нахождения расстояния Дамерау-Левенштейна в итеративной и рекурсивной формах. Помимо этого, были предоставлены результаты тестов на правильность данных реализаций, сравнительный анализ потребляемой памяти всех реализаций на слов длиной 4 и 1000. Было выявлено, что и при длине слов 4, и при длине слов 1000 итеративные реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна сравнимы по потребляемой памяти, причем разница между ними всегда 28 байт (3 и 0.0001 процента при длине слова 4 и 1000 соответственно). Рекурсивная реализация же при длине слов 4 потребляет на 37 процентов больше памяти, в то время как при длине слов 1000 она потребляет меньше памяти на 84 процента. По зависимостям потребляемой памяти от длин строк (рис. 5) мы видим, что рекурсивная реализация более затратна на промежутке от 1 до 59, что вызвано тем, что в функцию всегда передается копия строки, из-за того, что в python нет указателей, чтобы передавать их в функции вместо копий, иначе бы рекурсивная реализация была всегда эффективнее по памяти.

3 Экспериментальная часть

3.1 Постановка эксперимента

Должны быть произведены замеры времени работы каждого из алгоритмов при длинах строк от 2 до 9. Каждый тест должен быть проведен 100 раз, как результат должно быть взято среднее значение для уменьшения роли случайных факторов в итоге.

3.2 Результаты эксперимента

Ниже приведены графики со сравнением требуемого процессорного времени на итеративные алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна (рис 6) с длинами строк 2-9 и со сравнением времени, затрачиваемого на поиск расстояния Дамерау-Левенштейна в итеративном и рекурсивном вариантах. Lev - поиск расстояния Левенштейна в итеративной форме, DamLevI - поиск расстояния Дамерау-Левенштейна в итеративной форме, DamLevR - поиск расстояния Дамерау-Левенштейна в рекурсивной форме, на оси абсцисс откладывается длина строк, на оси ординат - затрачиваемое время в секундах.

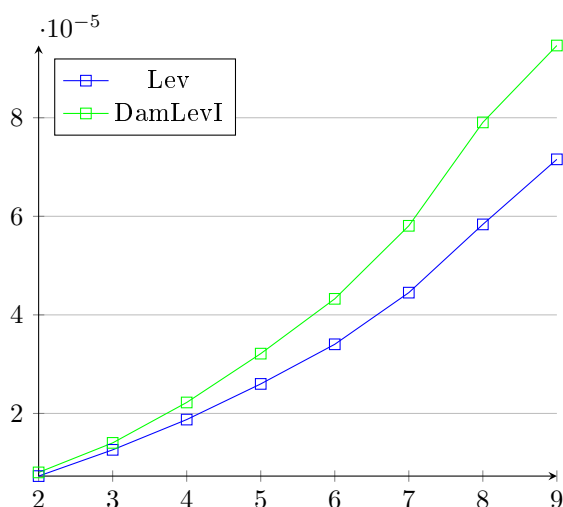


Рис. 6: сравнение времени на поиск Левенштейна и Дамерау-Левенштейна в итеративной форме

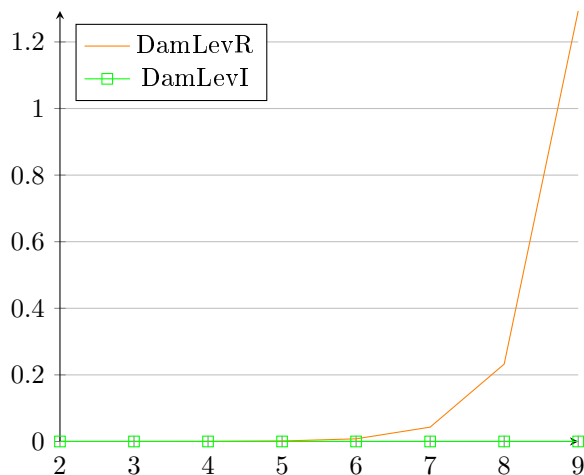


Рис. 7: сравнение времени на поиск Дамерау-Левенштейна в итеративной и рекурсивной формах

Таблица 11. Время, затрачиваемое различными алгоритмами на обработку строк длин от 2 до 9(в секундах).

Длина	Левенштейна	м. Дамерау-Левенштейна	р. Дамерау-Левенштейна
2	7.2e-06	8.0e-06	1.0e-05
3	1.2e-05	1.4e-05	5.7e-05
4	1.8e-05	2.2e-05	2.8e-04
5	2.5e-05	3.2e-05	1.4e-03
6	3.4e-05	4.3e-05	7.1e-03
7	4.4e-05	5.8e-05	4.3e-02
8	5.8e-05	7.9e-05	0.2
9	7.1e-05	9.4e-05	1.3

3.3 Вывод

По первому графику видно, что временные затраты на итеративный алгоритмы Левенштейна и Дамерау-Левенштейна сравнимы, но при этом алгоритм Дамерау-Левенштейна всегда медленнее. Из второго графика мы замечаем то, что рекурсивный алгоритм Дамерау-Левенштейна на порядки более затратный по времени, чем итеративный, начиная с длины строки в 5.

Заключение

В ходе данной лабораторной работы мною были реализованы алгоритмы Левенштейна в матричной форме и Дамерау-Левенштейна в матричной и рекурсивной форме. В ходе проверки на временные затраты было выявлено, что матричные реализации алгоритма Левенштейна и Дамерау-Левенштейна сравнимы по затрачиваемым процессорным ресурсам при длинах слов от 2 до 9 (разница между ними растет от 11 до 25 процентов в пользу Левенштейна). По используемой памяти разница меньше: от 0 до 3 процентов. Также было выявлено, что начиная со строк длиной в 3 символа рекурсивный вариант Дамерау-Левенштейна на порядки более затратный по процессорному времени, чем матричный (от 12.6 до 1.3×10^4 раз). Это вызвано тем, что в рекурсивном виде алгоритм одни и те же расчеты производит по несколько раз, так как расстояние между ними и теми же промежуточными словами может быть запрошено в нескольких независимо вызванных функциях, в то время как в матричном варианте все промежуточные расчеты записываются в матрицу и не пересчитываются. Говоря о памяти: рекурсивная форма нахождения расстояния Дамерау-Левенштейна при длине строки в 1000 на 84 процента меньше памяти, хотя при длине строки 4 потребляет на 34 процента больше памяти. Рекурсивная форма потребляет больше памяти на интервале от 1 до 59, при всех остальных значениях длин строк итеративная реализация более затратна по памяти.

Список литературы

- [1] Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход.-М.:Техносфера, 2009.
- [2] Нечёткий поиск в тексте и словаре // [Электронный ресурс]. Режим доступа: <https://habr.com/ru/post/114997/> (дата обращения: 2.10.19).
- [3] Нечеткий поиск, расстояние левенштейна алгоритм // [Электронный ресурс]. Режим доступа: <https://steposleep.ru/antanarivo-106/> (дата обращения: 2.10.19).