



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИУ, Информатика и системы управления

КАФЕДРА ИУ7, Программное обеспечение ЭВМ и информационные технологии

ЛАБОРАТОРНАЯ РАБОТА №1

ПО ДИСЦИПЛИНЕ

“Анализ алгоритмов”

Студент ИУ7-54Б
 (Группа)
 (И.О.Фамилия)

(Подпись, дата)

Преподаватель

(Подпись, дата)

(И.О.Фамилия)

2021 г.

Оглавление

Введение.	3
Аналитическая часть	4
Матричный алгоритм Левенштейна	4
Рекурсивный алгоритм Левенштейна	4
Рекурсивный алгоритм Левенштейна с заполнением матрицы	4
Алгоритм Дамерау-Левенштейна	5
Выводы из аналитического раздела	5
Конструкторская часть.	6
Схемы алгоритмов	6
Вывод	6
Технологическая часть.	7
Требования к программному обеспечению	7
Выбор и обоснование языка и среды программирования.	7
Реализация алгоритмов	7
Тестовые данные	12
Вывод	12
Исследовательская часть.	12
Демонстрация работы программы	12
Технические характеристики	13
Время выполнения алгоритмов	13
Использование памяти	14
Вывод	14
Заключение.	15
Список использованной литературы	16

Введение.

Данная лабораторная работа посвящена изучению алгоритмов нахождения расстояния Левенштейна и Дamerau-Левенштейна.

Расстояние Левенштейна - это минимальное количество операций вставки/удаления/замены одного символа на другой, необходимых для превращения одной строки в другую.

Вычисление расстояния Левенштейна применяется во многих отраслях для исправления ошибок в слове в компьютерной лингвистике, для сравнения хромосом и белков в биоинформатике.

Цель данной лабораторной работы: Изучение и оценка реализации методов динамического программирования на нахождение расстояния Левенштейна и Дamerau-Левенштейна.

Задачи данной лабораторной работы:

1. Изучение алгоритмов Левенштейна и Дamerau-Левенштейна;
2. Получение навыков реализации матричных и рекурсивных версий алгоритмов;
3. Проведение сравнительного анализа линейной и рекурсивной реализации реализаций алгоритмов;
4. Формирование обоснования полученных результатов исследования работы алгоритмов

1 Аналитическая часть

Расстояние Левенштейна - это минимальное количество операций вставки/удаления/замены одного символа на другой, необходимых для превращения одной строки в другую. Каждая операция определяется своей ценой, в общем случае операции определены, как:

- $\omega(\lambda, b)$ - цена операции вставки
- $\omega(a, \lambda)$ - цена операции удаления
- $\omega(a, b)$ - цена операции замены

Расстояние Левенштейна - это минимальная суммарная цена после последовательности замен. Существуют частные случаи нахождения расстояния Левенштейна:

- $\omega(a, a) = 0$
- $\omega(\lambda, b) = 1$
- $\omega(a, \lambda) = 1$
- $\omega(a, b) = 1$ и $a \neq b$

1.1 Матричный алгоритм Левенштейна

Данный метод реализации алгоритма Левенштейна эффективнее, чем рекурсивный, так как промежуточные значения хранятся в виде матрицы, при этом в любой момент времени мы можем обратиться при помощи индексации по строке и столбцу к любой из ранее выполненных операций. В общем виде алгоритм выглядит, как построчное заполнение матрицы:

$$A_{i,j} = D(i, j)$$

1.2 Рекурсивный алгоритм Левенштейна

Формула (1.1) расстояния между двумя строками a и b , где $|a|$ — длина строки a , $a[i]$ - i -й символ строки a :

$$D(i, j) = \begin{cases} 0 & i = 0, j = 0 \\ i & j = 0, i > 0 \\ j & i = 0, j > 0 \\ \min\{ & \\ D(i, j - 1) + 1 & \\ D(i - 1, j) + 1 & i > 0, j > 0 \\ D(i - 1, j - 1) + m(a[i], b[j]) & (1.2) \\ \} & \end{cases}, \quad (1.1)$$

$$m(a, b) = \begin{cases} 0 & \text{если } a = b, \\ 1 & \text{иначе} \end{cases}. \quad (1.2)$$

1.3 Рекурсивный алгоритм Левенштейна с заполнением матрицы

Рекурсивный алгоритм Левенштейна с заполнением матрицы - это объединение алгоритмов 1.1. и 1.2.: во время реализации рекурсивного алгоритма Левенштейна происходит заполнение матрицы. Ранее найденные расстояния не рассчитываются заново, они берутся из матрицы.

1.4 Алгоритм Дамерау-Левенштейна

Формула (1.3) нахождения расстояния Дамерау-Левенштейна определяется так, как и (1.1) - формула неэффективна по времени и аналогично для оптимизации используется добавление матрицы для хранения промежуточных значений.

$$d_{a,b}(i, j) = \begin{cases} \max(i, j), & \text{если } \min(i, j) = 0, \\ \min\{ & \\ d_{a,b}(i, j - 1) + 1, & \\ d_{a,b}(i - 1, j) + 1, & \\ d_{a,b}(i - 1, j - 1) + m(a[i], b[j]), & \text{иначе} \\ \left[\begin{array}{ll} d_{a,b}(i - 2, j - 2) + 1, & \text{если } i, j > 1; \\ & a[i] = b[j - 1]; \\ & b[j] = a[i - 1] \\ & \infty, \quad \text{иначе} \end{array} \right. & \\ \} & \end{cases}, \quad (1.3)$$

1.5 Выводы из аналитического раздела

В данном разделе были описаны: рекурсивный алгоритм Левенштейна с матрицей и без нее, матричный итерационный алгоритм Левенштейна, итерационный алгоритм Дамерау-Левенштейна.

2 Конструкторская часть.

В данном разделе будут приведены блок-схемы алгоритмов, описанных в аналитическом разделе п.1.

2.1 Схемы алгоритмов

2.2 Вывод

Блок-схемы в данном разделе позволяют перейти к технологической части - непосредственно к программной реализации решения.

3 Технологическая часть.

В данном разделе будут рассмотрены требования к разрабатываемому программному обеспечению, средства, использованные в процессе разработки для реализации поставленных задач.

3.1 Требования к программному обеспечению

Программное обеспечение должно реализовывать поставленную на лабораторную работу задачу. Интерфейс для взаимодействия с программой - командная строка. Программа должна выводить полученное расстояние между двумя введенными строками и показывать потраченное на это время.

3.2 Выбор и обоснование языка и среды программирования.

Для разработки данной программы применён язык Python 3 с библиотекой `time.clock()` для вычисления времени работы процессора, потому что я хочу расширить свои знания в области данного языка программирования.

3.3 Реализация алгоритмов

В листингах 1-4 приведена реализация алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Программа была реализована в парадигме ООП, где в базовый класс был вынесен объект `Levenstein` (Листинг 1.), внутри него с доступом `protected`, используемая алгоритмами с меморизацией и без нее, рекурсивная функция получения расстояния между строками.

Наследуемые объекты рекурсивного алгоритма без кэша (Листинг 2), рекурсивного алгоритма с кэшем (Листинг 3), итерационный алгоритм Дамерау-Левенштейна (Листинг 4), итерационный алгоритм Левенштейна (Листинг 5) имеют публичную функцию получения времени работы `get_time()`;

Листинг 1: Базовый класс `Levenstein` с Рекурсивная функция нахождения расстояния Левенштейна

```
1. # Объект алгоритма Левенштейна
2. class Levenshtein:
3.     # Защищенные наследуемые данные объекта
4.     _first_string = None
5.     _second_string = None
6.
7.     # Ключевое расстояние
8.     _distance = None
9.
10.    # Ключевое время
11.    _time = None
12.
```

```

13.     # Создание объекта
14.     def __init__(self, first_string, second_string):
15.         # Назначение данных объекта
16.         self._first_string = first_string
17.         self._second_string = second_string
18.
19.         # Установка значения расстояния
20.         self._distance = config.START_ZERO_VALUE
21.
22.         # Установка значения времени выполнения
23.         self._time = config.START_ZERO_VALUE
24.
25.     # Общая функция получения расстояния между двумя строками
26.     def get_distance(self):
27.         return self._distance
28.
29.     # Общая функция получения времени выполнения
30.     def get_time(self):
31.         return self._time
32.
33.     # Получение расстояния между строками
34.     def _recursive_get_distance(self, first_string_length,
35. second_string_length):
36.         # если одна из строк пустая, то расстояние до другой
37.         строки - ее длина
38.         # т.е. n вставок
39.         if first_string_length == 0 or second_string_length
40. == 0:
41.             return max(first_string_length,
42. second_string_length)
43.
44.         # если оба последних символов одинаковые, то съедаем
45.         их оба, не меняя расстояние
46.         elif self._first_string[first_string_length - 1] ==
47. self._second_string[second_string_length - 1]:
48.             return
49. self._recursive_get_distance(first_string_length - 1,
50. second_string_length - 1)
51.
52.         # выбор минимального значения из трех
53.         else:
54.             return 1 + min(
55. self._recursive_get_distance(first_string_length,
56. second_string_length - 1), # Удаление
57. self._recursive_get_distance(first_string_length - 1,
58. second_string_length), # Вставка
59. self._recursive_get_distance(first_string_length - 1,
60. second_string_length - 1) # Замена
61. )

```

Листинг 2: Наследуемый класс Левенштейна без кэша


```

1. # Наследуемый объект Рекурсивного алгоритма без кэша
2. class LevenshteinRecursiveWithoutCache(Levenshtein):
3.
4.     # Общая функция получения расстояния между двумя строками
5.     def get_distance(self):
6.         self._distance =
7.             self._recursive_get_distance(len(self._first_string),
8.             len(self._second_string))
9.         return self._distance
10.
11.     # Получение времени
12.     def get_time(self):
13.         t_0 = clock()
14.         self._recursive_get_distance(len(self._first_string),
15.         len(self._second_string))
16.         t_1 = clock()
17.         self._time = t_1 - t_0
18.         return self._time

```

Листинг 3: Наследуемый класс Левенштейна с кешем

```

1. # Наследуемый объект Рекурсивного алгоритма с кэшем
2. class LevenshteinRecursiveWithCache(Levenshtein):
3.     _first_string_length = None
4.     _second_string_length = None
5.
6.     def __init__(self, first_string, second_string):
7.         super().__init__(first_string, second_string)
8.
9.         self._first_string_length = len(self._first_string)
10.        self._second_string_length = len(self._second_string)
11.
12.        # Получение времени
13.        def get_time(self):
14.            t_0 = clock()
15.            self.get_distance()
16.            t_1 = clock()
17.
18.            self._time = t_1 - t_0
19.
20.            return self._time
21.
22.        def get_distance(self):
23.            # Общая функция получения расстояния между двумя
24.            # строками
25.            @lru_cache(maxsize=self._first_string_length *
26.            self._second_string_length)
27.            def get_distance():
28.                self._distance =
29.                    self._recursive_get_distance(len(self._first_string),
30.                    len(self._second_string))
31.                return self._distance

```

```

28.
29.         # Обновление расстояния
30.         self._distance = get_distance()
31.
32.         return self._distance

```

Листинг 4: Наследуемый класс итерационного Дамерау Левенштейна

```

1. # Объект вычисления Дамерау Левенштейна
2. class DamerauLevenshtein(Levenshtein):
3.     # Используемые блины строк
4.     _first_string_length = None
5.     _second_string_length = None
6.
7.     def __init__(self, first_string, second_string):
8.         super().__init__(first_string, second_string)
9.
10.        self._first_string_length = len(self._first_string)
11.        self._second_string_length = len(self._second_string)
12.
13.        # Получение расстояния между двумя строками
14.        def get_distance(self):
15.            d = {}
16.
17.            for i in range(-1, self._first_string_length + 1):
18.                d[(i, -1)] = i + 1
19.            for j in range(-1, self._second_string_length + 1):
20.                d[(-1, j)] = j + 1
21.
22.            for i in range(self._first_string_length):
23.                for j in range(self._second_string_length):
24.                    if self._first_string[i] ==
self._second_string[j]:
25.                        cost = 0
26.                    else:
27.                        cost = 1
28.                    d[(i, j)] = min(
29.                        d[(i - 1, j)] + 1, # deletion
30.                        d[(i, j - 1)] + 1, # insertion
31.                        d[(i - 1, j - 1)] + cost, # substitution
32.                    )
33.                    if i and j and self._first_string[i] ==
self._second_string[j - 1] and \
34.                        self._first_string[i - 1] ==
self._second_string[j]:
35.                        d[(i, j)] = min(d[(i, j)], d[i - 2, j - 2]
+ cost) # transposition
36.
37.            return d[self._first_string_length - 1,
self._second_string_length - 1]
38.
39.        # Получение времени
40.        def get_time(self):
41.            t_0 = clock()
42.            self.get_distance()

```

```

43.         t_1 = clock()
44.
45.         self._time = t_1 - t_0
46.
47.         return self._time

```

Листинг 5: Наследуемый класс Левенштейна с кешем

```

1. # Линейный алгоритм вычисления Левенштейна
2. class LevenshteinLinear(Levenshtein):
3.     # Используемые длины строк
4.     _first_string_length = None
5.     _second_string_length = None
6.
7.     def __init__(self, first_string, second_string):
8.         super().__init__(first_string, second_string)
9.         self._first_string_length = len(first_string)
10.        self._second_string_length = len(second_string)
11.
12.        def _update_distance(self):
13.            if self._first_string_length >
self._second_string_length:
14.                self._first_string, self._second_string =
self._second_string, self._first_string
15.                self._first_string_length,
self._second_string_length = self._second_string_length,
self._first_string_length
16.
17.                current_row = range(self._first_string_length + 1)
18.                for i in range(1, self._second_string_length + 1):
19.                    previous_row, current_row = current_row, [i] + [0]
* self._first_string_length
20.                    for j in range(1, self._first_string_length + 1):
21.                        add, delete, change = previous_row[j] + 1,
current_row[j - 1] + 1, previous_row[j - 1]
22.                        if self._first_string[j - 1] !=
self._second_string[i - 1]:
23.                            change += 1
24.                            current_row[j] = min(add, delete, change)
25.
26.                return current_row[self._first_string_length]
27.
28.        # Получение расстояния между двумя строками
29.        def get_distance(self):
30.            self._distance = self._update_distance()
31.            return self._distance
32.
33.        # Получение времени
34.        def get_time(self):
35.            t_0 = clock()
36.            self.get_distance()
37.            t_1 = clock()
38.
39.            self._time = t_1 - t_0
40.

```

```
41.         return self._time
```

3.4 Тестовые данные

Тестовые данные, на которых было протестировано разработанное программное обеспечение, представлено в Таблице 1.

Таблица 1: Тестовые данные

№	Первое слово	Второе слово	Ожидаемый результат				Полученный результат			
			Расстояние				Расстояние			
			Л. 2	Л. 3	Л. 4	Л.5	Л. 2	Л. 3	Л. 4	Л. 5
1	увлечение	развлечения	4	4	4	4	4	4	4	4
2	кот	скат	2	2	2	2	2	2	2	2
3		тест	4	4	4	4	4	4	4	4
4	мгту	мтгу	2	2	1	2	2	2	1	2
5	рот	кот	1	1	1	1	1	1	1	1
7	лилия	рим	4	4	4	4	4	4	4	4
8	рим	мир	2	2	2	2	2	2	2	2
9	apple	aple	2	2	1	2	2	2	1	2
10	катя	надя	2	2	2	2	2	2	2	2

3.5 Вывод

В аналитическом разделе были представлены разработанный код, демонстрация его работы на тестовых данных Таблицы 1.

4 Исследовательская часть.

4.1. Демонстрация работы программы

Пример работы программы представлен на рисунке 1.

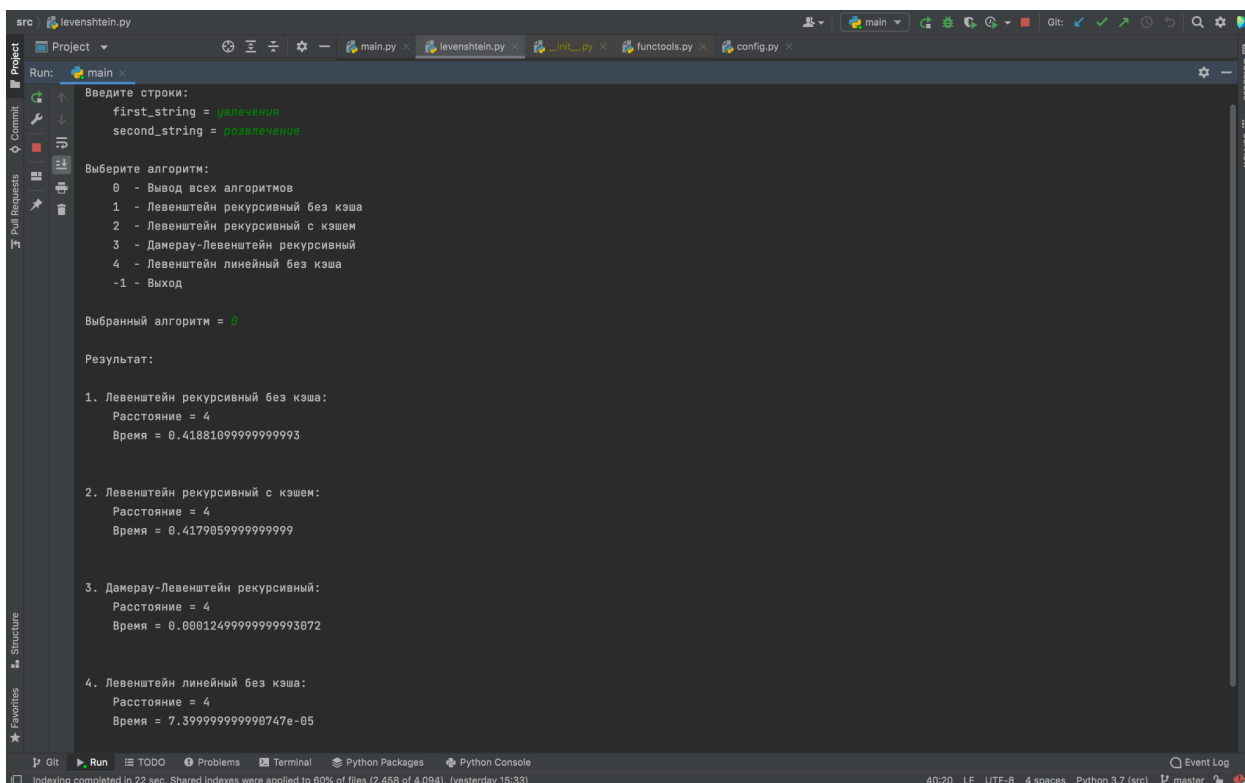


Рисунок 1: Демонстрация работы программы на примере строк Увлечение и Развлечение

4.2. Технические характеристики

В Таблице 2. приведены технические характеристики ЭВМ, на котором проводилось тестирование разрабатываемого программного обеспечения.

Таблица 2: Технические характеристики ЭВМ, на котором проводилось тестирование разрабатываемого программного обеспечения

ОС	Mac OS Mojave 64-bit
ОЗУ	8 Gb 2133 MHz LPDDR3
Процессор	2,3 GHz Intel Core i5

4.3. Время выполнения алгоритмов

Таблица 3: Таблица времени выполнения алгоритмов (в наносекундах)

№	Длина строк	Время			
		Л.1	Л.2	Л.3	Л.3
1	10	2.4e-05	3.8e-05	0.0005	0.0003
2	20	3.7e-05	5.09e-05	0.0009	0.001
3	40	3.9e-05	5.49e-05	0.003	0.001

4	100	8.9e-05	9.8e-05	0.024	0.013
5	200	0.0001	0.00012	0.0204	0.0111

4.4. Использование памяти

В Таблице 4 представлена информация об использовании памяти во время выполнения разных типов алгоритмов.

Таблица 3: Таблица времени выполнения алгоритмов (в наносекундах)

№	Тип вызова	Память
1	Рекур-ый вызов	$(S(STR_1) + S(STR_2)) \cdot (2 \cdot S(string) + 3 \cdot S(integer))$
2	Итер-ый вызов	$(S(STR_1) + 1) \cdot (S(STR_2) + 1) \cdot S(integer) + 5 \cdot S(integer) + 2 \cdot S(string)$

4.5. Вывод

Время работы рекурсивной версии алгоритма увеличивается в геометрической прогрессии, это самый неэффективный способ реализации алгоритма нахождения расстояния Левенштейна по времени и памяти. Наиболее эффективный способ из представленных - итеративный с хранением двух строк.

Заключение.

В данной лабораторная работа я изучил изучению алгоритмы нахождения расстояния Левенштейна и Дamerau-Левенштейна, провел Изучение и оценку реализации методов динамического программирования на нахождение расстояния Левенштейна и Дamerau-Левенштейна, получил навыки реализации матричных и рекурсивных версий алгоритмов, провел сравнительный анализ линейной и рекурсивной реализации алгоритмов, сформировал обоснования полученных результатов исследования алгоритмов.

Список использованной литературы

1. Роджерс Д., Алгоритмические основы машинной графики: пер. с англ.— М.: Мир, 1989.— 512 с.: ил.
2. Марк Пилгрим, Dive into Python, 2017.
3. Steven F. Lott , Dusty Phillips, Python Object-Oriented Programming - Fourth Edition
4. Проблемы трассировки лучей – из будущего в реальное время. [Электронный ресурс]. – Режим доступа: <https://nvworld.ru/articles/ray-tracing/3/>
5. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж., Приёмы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.: ил. (Серия «Библиотека программиста»)
6. Particle Systems. [Электронный ресурс]. – Режим доступа: <http://homepages.inf.ed.ac.uk/tkomura>
7. А.Н.Романюк, М.В.Куринный, АЛГОРИТМЫ ПОСТРОЕНИЯ ТЕНЕЙ