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 «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по Лабораторной работе

по курсу «Анализ Алгоритмов»

на тему: «Расстояние Левенштейна»

Студент ИУ7-53Б _____ Миронов Г. А.
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Волкова Л. Л.
(Подпись, дата) (И. О. Фамилия)

2021 г.

СОДЕРЖАНИЕ

Введение **3**

1 Аналитическая часть **5**

1.1 Некоторые теоретические сведения 5

1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна . 5

1.3 Итеративный алгоритм нахождения расстояния Левенштейна с кэшированием 7

1.4 Расстояние Дамерау-Левенштейна 8

1.5 Вывод 9

2 Конструкторская часть **10**

2.1 Разработка алгоритмов 10

2.2 Вывод 14

3 Технологическая часть **15**

3.1 Выбор языка программирования 15

3.2 Требования к программному обеспечению 15

3.3 Сведения о модулях программы 15

3.4 Вывод 20

4 Экспериментальная часть **21**

4.1 Тестирование 21

4.2 Технические характеристики 21

4.3 Временные характеристики 22

4.4 Характеристики по памяти 24

4.5 Сравнительный анализ алгоритмов 25

4.6 Вывод 25

Заключение **26**

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ **27**

Введение

В данной лабораторной работе будет рассмотрено расстояние Левенштейна.

Расстояние Левенштейна [1] (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов. Она определяется как минимальное количество односимвольных операций (а именно вставки, удаления, замены), необходимых для превращения одной последовательности символов в другую. В общем случае, операциям, используемым в этом преобразовании, можно назначить разные цены.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0–1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Расстояние Левенштейна и его обобщения активно применяются:

1) для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

2) в биоинформатике для сравнения генов, хромосом и белков.

3) для сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы);

Расстояние Дамерау — Левенштейна (названо в честь учёных Фредерика Дамерау и Владимира Левенштейна) — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов.

Целью данной работы является реализация и изучение следующих алгоритмов:

- Рекурсивный алгоритм нахождения расстояния Левенштейна.

- Рекурсивный алгоритм нахождения расстояния Дамерау–Левенштейна.
- Итеративный алгоритм нахождения расстояния Левенштейна с кэшированием.
- Итеративный алгоритм нахождения расстояния Дамерау–Левенштейна с
  кэшированием.

Для достижения данной цели необходимо решить следующие задачи:

1. изучить расстояния Левенштейна и Дамерау–Левенштейна;
2. реализовать алгоритмы поиска расстояний: нерекурсивный (с кэшем) и рекурсивные алгоритмы поиска расстояния Левенштейна и расстояния Дамерау–Левенштейна;
3. замерить процессорное время работы реализаций алгоритмов поиска расстояний;
4. сравнить временные характеристики, а также затраченную память;

1 Аналитическая часть

1.1 Некоторые теоретические сведения

При преобразовании одного слова в другое можно использовать следующие операции.

1. D (от англ. delete) - удаление;
2. I (от англ. insert) - вставка;
3. R (от англ replace) - замена;

Будем считать стоимость каждой вышеизложенной операции равной 1.

Введем понятие совпадения - M (от англ. match). Его стоимость будет равна 0.

1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна

Расстояние Левенштейна между двумя строками a и b может быть вычислено по формуле 1.1, где $|a|$ означает длину строки a; $a[i]$ — i-ый символ строки a , функция $D(i, j)$ определена как:

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

а функция 1.2 определена как:

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

Рекурсивный алгоритм реализует формулу 1.1. Функция 1.1 составлена из следующих соображений.

1. Для перевода из пустой строки в пустую требуется ноль операций.
2. Для перевода из пустой строки в строку a требуется $|a|$ операций.
3. Для перевода из строки a в пустую требуется $|a|$ операций.
4. Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность проведения любых двух операций можно поменять, порядок проведения операций не имеет никакого значения. Полагая, что a', b' – строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:
   - сумма цены преобразования строки a в b и цены проведения операции удаления, которая необходима для преобразования a' в a;
   - сумма цены преобразования строки a в b и цены проведения операции вставки, которая необходима для преобразования b' в b;
   - сумма цены преобразования из a' в b' и операции замены, предполагая, что a' и b' оканчиваются разные символы;
   - цена преобразования из a' в v' , предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.3 Итеративный алгоритм нахождения расстояния Левенштейна с кэшированием

Прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения при больших i, j, т. к. множество промежуточных значений $D(i, j)$ вычисляются заново множество раз подряд. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях

хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i,j)$.

Однака, матричный алгоритм является малоэффективным по памяти при больших i, j, т. к. множество промежуточных значений $D(i,j)$ хранится в памяти после их использования. Для оптимизации нахложднения расстояния Левенштейна можно использовать кэш, т.е. пару строк, содержащюю значения $D(i,j)$, вычисленные в предыдущей итерации алгоритма и значения $D(i,j)$, вычисляемые в текущей итерации.

1.4 Расстояние Дамерау-Левенштейна

В алгоритме поиска расстояния Дамерау-Левенштейна, помимо 3 операций (удаление, вставка, замена), присутсвующих в алгоритме поиска расстояния Левенштейна, задействуется еще одна редакторская операция - транспозиция T (от англ. transposition). Расстояние Дамерау-Левенштейна рассчитывается по рекуррентной формуле 1.3.

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

1.5 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов. Формулы Левенштейна и Дамерау – Левенштейна для рассчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

В данном разделе будут рассмотрены схемы вышеизложенных алгоритмов.

2.1 Разработка алгоритмов

На рисунке 2.1 представлен рекурсивный алгоритм поиска расстояния Левенштейна.

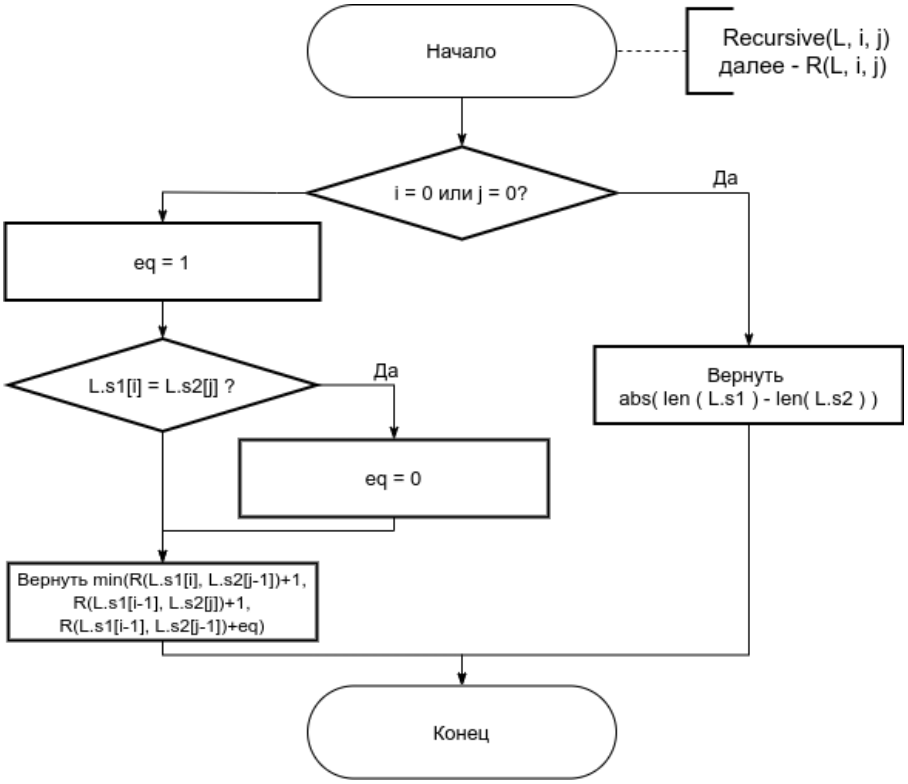

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

На рисунке 2.2 представлен алгоритм поиска расстояния Левенштейна с кэшированием.

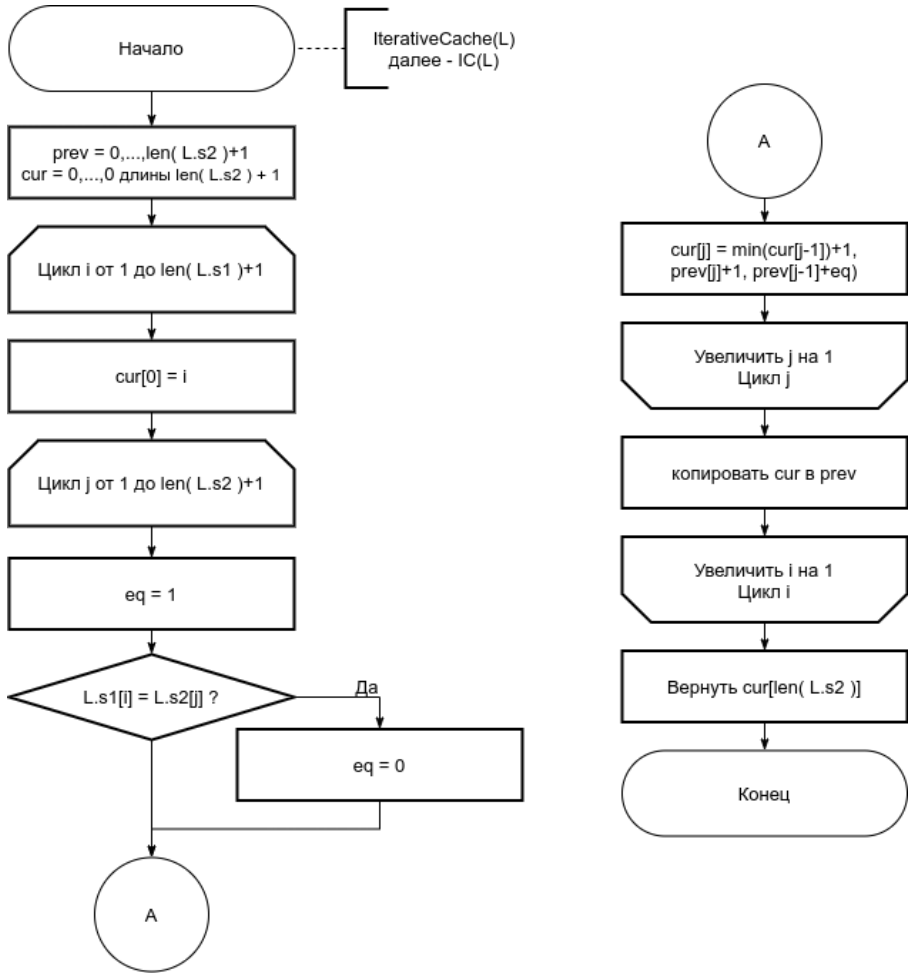

Рисунок 2.2 – Схема алгоритма нахождения расстояния Левенштейна с кеширование

На рисунке 2.3 представлен рекурсивный алгоритм поиска расстояния Дамерау – Левенштейна.

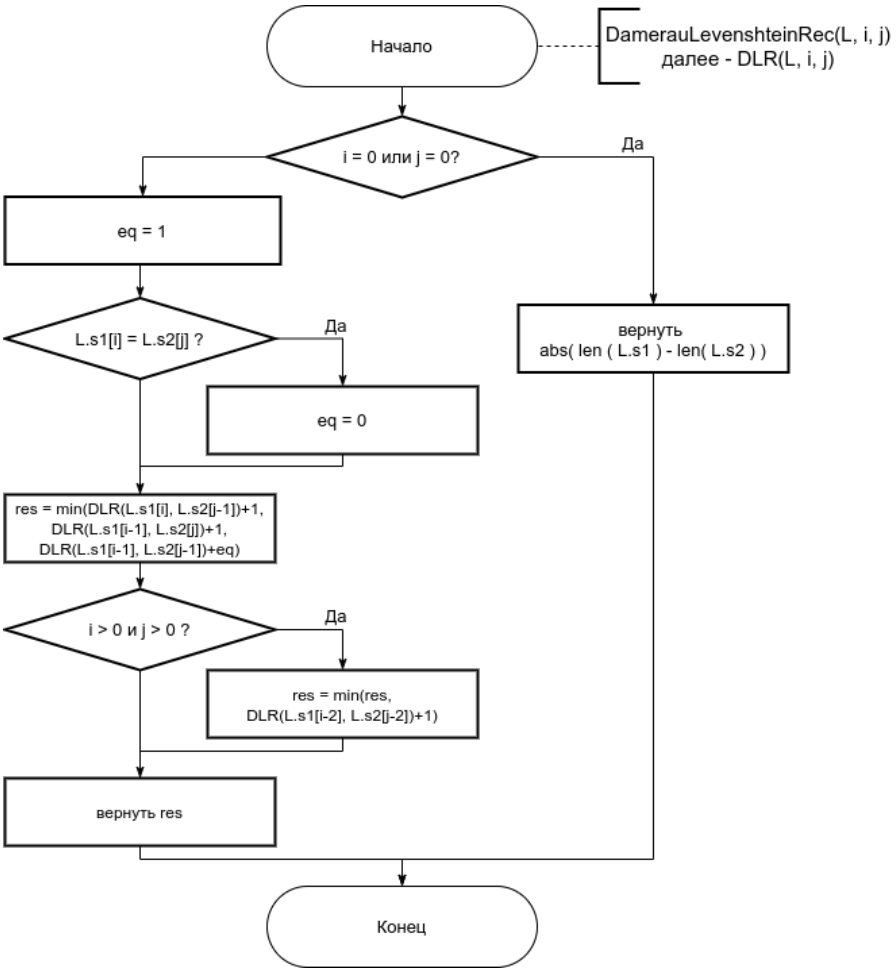

Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Дамерау – Левенштейна

На рисунке 2.4 представлен алгоритм поиска расстояния Дамерау – Левенштейна с кэшированием.

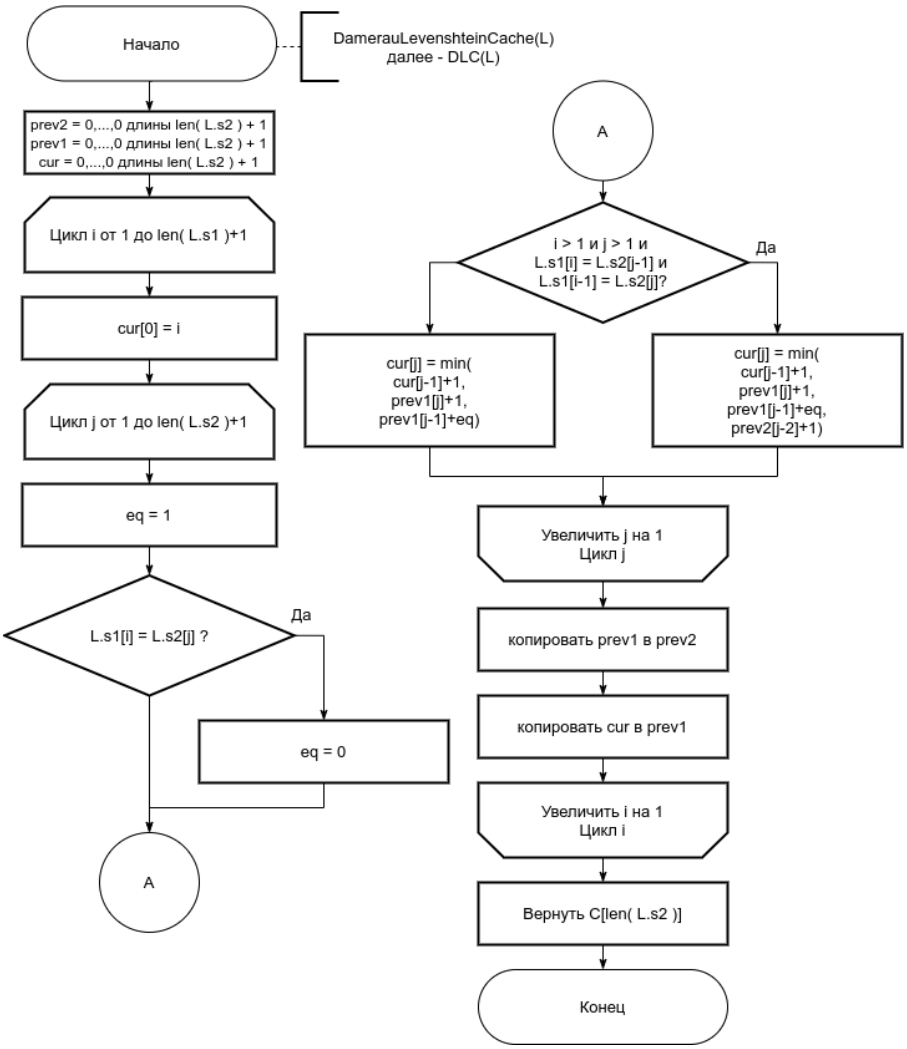

Рисунок 2.4 – Схема алгоритма нахождения расстояния Дамерау – Левенштейна с кэшированием

2.2 Вывод

На основе теоретических данных, полученных из аналитического раздела были построены схемы требуемых алгоритмов.

3 Технологическая часть

В данном разделе приведены требования к программному обеспечению, средства реализации и листинги кода.

3.1 Выбор языка программирования

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Golang [2]. Данный выбор обусловлен тем, что я имею некоторый опыт разработки на нем, а так же наличием у языка встроенных высокоточных средств тестирования и анализа разработанного ПО.

3.2 Требования к программному обеспечению

К программе предъявляется ряд требований:

- На вход подаётся две регистрозависимых строки.
- На выходе — результат выполнения каждого из вышеуказанных алгоритмов.

3.3 Сведения о модулях программы

Данная программа разбита на следующие модули:

- `main.go` – Файл, содержащий точку входа в программу. В нем происходит общение с пользователем и вызов алгоритмов.
- `io.go` – Файл содержит функции для вввода и вывода информации.
- `iterative_cash.go` – Файл содержит реализацию поиска расстояния Левенштейна с кэшированием.
- `iterative_dl_cash.go` – Файл содержит реализацию поиска расстояния Дамерау-Левенштейна с кэшированием.
- `recursive_dl.go` – Файл содержит рекурсивную реализацию поиска расстояния Дамерау-Левенштейна.
- `recursive.go` – Файл содержит рекурсивную реализацию поиска расстояния Левенштейна.

- `types.go` – Файл содержит пользовательские типы данных, используемые в алгоритмах.

- `utils.go` – Файл содержит различные функции для вычислений.

В листингах 3.4–3.7 представлены исходные коды разобранных ранее алгоритмов.

Листинг 3.1 – Основной файл программы main

```
1  func main() {
2      fmt.Println(aurora.Magenta("Levinstein␣distance\n"))
3
4      fst, snd := getWords()
5
6      printMethodRes("Recursive␣method",
7          levenshtein.NewLevenstein(fst, snd).Recursive())
8
9      printMethodRes("Recursive␣Damerau-Levenstein␣method",
10         levenshtein.NewLevenstein(fst, snd).DamerauLevenshteinRec
               ())
11
12     printMethodRes("Iterative␣method␣with␣cash",
13         levenshtein.NewLevenstein(fst, snd).IterativeCache())
14
15     printMethodRes("Damerau-Levenstein␣with␣cash",
16         levenshtein.NewLevenstein(fst, snd).
               DamerauLevenshteinCache())
17 }
```

Листинг 3.2 – Определение пользовательских типов данных

```
1  func NewLevenstein(s1, s2 string) Levenshtein {
2      return Levenshtein{[]rune(s1), []rune(s2)}
3  }
4
5  type Levenshtein struct {
6      s1 []rune
7      s2 []rune
8  }
```

Листинг 3.3 – Различные функции для вычислений

```
func minFromThree(a, b, c int) int {
    return minFromTwo(a, minFromTwo(b, c))
}

func minFromTwo(a, b int) int {
    if (a < b) {
        return a
    }
    return b
}

func abs(x int) int {
    if x < 0 {
        return -x
    }
    return x
}
```

Листинг 3.4 – Рекурсивная функция нахождения расстояния Левенштейна

```
func (l Levenshtein) Recursive() int {
    return l.getDistance(len(l.s1), len(l.s2))
}

func (l *Levenshtein) getDistance(i, j int) int {
    if i == 0 || j == 0 {
        return abs(len(l.s1) - len(l.s2))
    }
    eq := 1
    if l.s1[i-1] == l.s2[j-1] {
        eq = 0
    }
    return minFromThree(
        l.getDistance(i, j-1)+1,
        l.getDistance(i-1, j)+1,
        l.getDistance(i-1, j-1)+eq)
}
```

Листинг 3.5 – Функция нахождения расстояния Левенштейна с кешем

```
func (l Levenshtein) IterativeCache() int {
    prev := make([]int, len(l.s2)+1)
    for i := 0; i <= len(l.s2); i++ {
        prev[i] = i
    }
    cur := make([]int, len(l.s2)+1)
    for i := 1; i <= len(l.s1); i++ {
        cur[0] = i
        for j := 1; j <= len(l.s2); j++ {
            eq := 1
            if l.s1[i-1] == l.s2[j-1] {
                eq = 0
            }
            cur[j] = minFromThree(
                cur[j-1]+1,
                prev[j]+1,
                prev[j-1]+eq,
            )
        }
        copy(prev, cur)
    }
    return cur[len(l.s2)]
}
```

Листинг 3.6 – Рекурсивная функция нахождения расстояния Дамерау–Левенштейна

```
1  func (l *Levenshtein) getDamerauLevenshteinRec(i, j int) int {
2      if i == 0 || j == 0 {
3          return abs(len(l.s1) - len(l.s2))
4      }
5      eq := 1
6      if l.s1[i-1] == l.s2[j-1] {
7          eq = 0
8      }
9      res := minFromThree(
10         l.getDamerauLevenshteinRec(i, j-1)+1,
11         l.getDamerauLevenshteinRec(i-1, j)+1,
12         l.getDamerauLevenshteinRec(i-1, j-1)+eq,
13     )
14     if i > 1 && j > 1 &&
15         l.s1[i-1] == l.s2[j-2] &&
16         l.s1[i-2] == l.s2[j-1] {
17         res = minFromTwo(res, l.getDamerauLevenshteinRec(i-2, j
               -2)+1)
18     }
19     return res
20 }
```

Листинг 3.7 – Функция нахождения расстояния Дамерау–Левенштейна с кэшем

```
func (l Levenshtein) DamerauLevenshteinCache() int {
    prev2 := make([]int, len(l.s2)+1)
    prev1 := make([]int, len(l.s2)+1)
    cur := make([]int, len(l.s2)+1)

    for i := 0; i <= len(l.s2); i++ {
        prev1[i] = i
    }

    for i := 1; i <= len(l.s1); i++ {
        cur[0] = i
        for j := 1; j <= len(l.s2); j++ {
            eq := 1
            if l.s1[i-1] == l.s2[j-1] {
                eq = 0
            }
            cur[j] = minFromThree(
                cur[j-1]+1,
                prev1[j]+1,
                prev1[j-1]+eq,
            )
            if i > 1 && j > 1 &&
                l.s1[i-1] == l.s2[j-2] &&
                l.s1[i-2] == l.s2[j-1] {
                cur[j] = minFromTwo(
                    cur[j],
                    prev2[j-2]+1,
                )
            }
        }
        copy(prev2, prev1)
        copy(prev1, cur)
    }

    return cur[len(l.s2)]
}
```

3.4 Вывод

Были реализованы алгоритмы: вычисления расстояния Левенштейна рекурсивно, с заполнением кэша, а также вычисления расстояния Дамерау–Левенштейна рекурсивно и вычисления расстояния Дамерау–Левенштейна с заполнением кэша.

4 Экспериментальная часть

В данном разделе будет проведено функциональное тестирование разработанного программного обеспечения. Так же будет произведено измерение временных характеристик и характеристик по памяти каждого из реализованных алгоритмов.

Для проведения подобных экспериментов на языке программирования `Golang` [2], используется специальный пакет `testing` [3]. Данный пакет предоставляет инструменты для измерения процессорного времени и объема памяти, использованных конкретным алгоритмом в ходе проведения эксперимента.

4.1 Тестирование

В таблице 4.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау–Левенштейна.

Таблица 4.1 – Функциональные тесты

Строка 1	**Строка 2**	**Ожидаемый результат**	
		Левенштейн	**Дамерау — Левенштейн**
		0	0
совпадение	совпадение	0	0
Кремль	Кремль	0	0
Ответвление	ответвление	1	1
exponential	polynomial	6	6
ура	уар	2	1
абв	ваб	2	2

При проведении функционального тестирования, полученные результаты работы программы совпали с ожидаемыми. Таким образом, функциональное тестирование пройдено успешно.

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

- Процессор: Intel Core™ i5-8250U [4] CPU @ 1.60GHz.
- Память: 32 GiB.

- Операционная система: Ubuntu [5] Linux [6] 20.04 64-bit.

Исследование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения рабочего стола, окружением рабочего стола, а также непосредственно системой тестирования.

4.3 Временные характеристики

Результаты замеров по результатам экспериментов приведены в Таблице 4.2. В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится " - ".

Таблица 4.2 – Замер времени для строк, размером от 5 до 200

	Время, нс			
	Левенштейн		**Дамерау – Левенштейн**	
Длина (символ)	**Рекурсивный**	**Итеративный**	**Рекурсивный**	**Итеративный**
1	13.03	49.73	13.63	73.86
2	59.30	65.18	61.58	96.58
3	291.0	85.21	303.6	125.6
4	1490	113.7	1553	168.3
5	7836	138.9	8193	206.4
10	37794683	356.1	40084588	562.5
15	206560303523	671.3	215290187860	1062
20	-	1156	-	1913
30	-	2409	-	4017
50	-	6908	-	12346
100	-	36683	-	59230
200	-	152706	-	228972

Отдельно сравним итеративные алгоритмы поиска расстояний Левенштейна и Дамерау–Левенштейна. Сравнение будет производится на основе данных, представленных в Таблице 4.2. Результат можно увидеть на Рисунке 4.1.

При длинах строк менее 30 символов разница по времени между итеративными реализациями незначительна, однако при увеличении длины строки алгоритм поиска расстояния Левенштейна оказывается быстрее вплоть до полутора раз (при длинах строк равных 200). Это обосновывается тем, что у алгоритма поиска расстояния Дамерау-Левенштейна задействуется дополнительная операция, которая замедляет алгоритм.

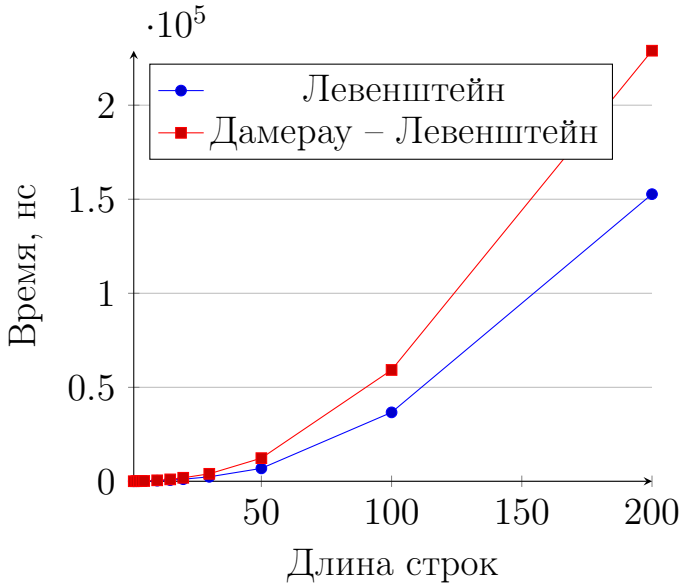

Рисунок 4.1 – Сравнение времени работы алгоритма поиска расстояния Левенштейна и Дамерау – Левенштейна

Так же сравним рекурсивную и итеративную реализации алгоритма поиска расстояния Левенштейна. Данные представлены в Таблице 4.2 и отображены на Рисунке 4.2.

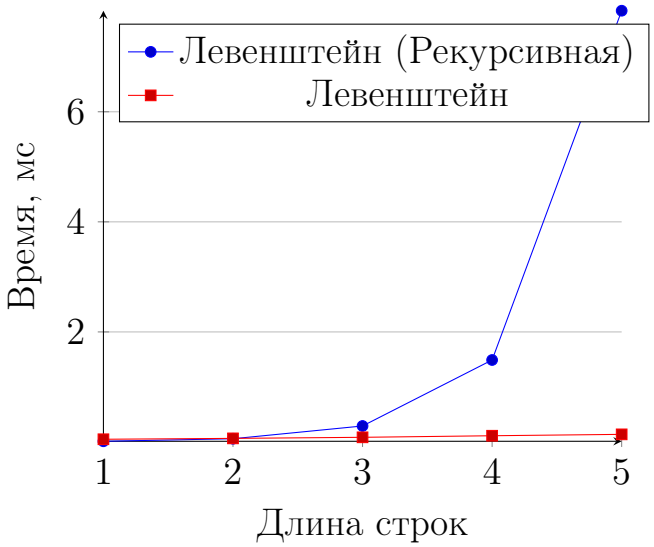

Рисунок 4.2 – Сравнение времени работы рекурсивной и итеративной кэширующей реализаций алгоритма Левенштейна.

На Рисунке 4.2 продемонстрировано, что рекурсивный алгоритм становится менее эффективным (вплоть до 56 раз при длине строк равной 5 элементов), чем итеративный.

Из этого можно сделать вывод о том, что при малых длинах строк (1–2 символа) предпочтительнее использовать рекурсивные алгоритмы,

однако при обработке более длинных строк (болеее 3 символов) итеративные алгоритмы оказываются многократно более эффективными и рекомендованы к использованию.

Из данных, приведенных в Таблице 4.2, видно, что итеративные алгоритмы становятся более эффективными по времени при увеличении длин строк, работая приблизительно в 308 млн. раз (Левенштейн) и 203 млн. раз (Дамерау – Левенштейн) быстрее, чем рекурсивные (при длинах строк равных 200). Однако, при малых длинах (1 – 2 элемента) рекурсивные алгоритмы являются более эффективными (вплоть до 3 раз), чем итеративные.

Кроме того, согласно данным, приведенным в Таблице 4.2, рекурсивные алгоритмы при длинах строк более 15 элементов не пригодны к использованию в силу экспоненциально роста затрат процессорного времени, в то время, как затраты итеративных алгоритмов по времени линейны.

4.4 Характеристики по памяти

Результаты замеров по результатам экспериментов приведены в Таблице 4.3. В данной таблице для значений, для которых тестирование не выполнялось, в поле результата находится " - ".

Таблица 4.3 – Замер расхода памяти для строк, размером от 5 до 200

	Память, байт			
	Левенштейн		**Дамерау – Левенштейн**	
Длина (символ)	**Рекурсивный**	**Итеративный**	**Рекурсивный**	**Итеративный**
1	0	32	0	48
2	0	48	0	72
3	0	64	0	96
4	0	96	0	144
5	0	96	0	144
10	0	192	0	288
15	0	256	0	384
20	-	352	-	528
30	-	512	-	768
50	-	832	-	1248
100	-	1792	-	2688
200	-	3584	-	5376

Из данных, приведенных в Таблице 4.3, видно, что рекурсивные алгоритмы являются более эффективными по памяти, так как не используют

дополнительной памяти в своей работе, в то время как итеративные алгоритмы затрачивают память линейно пропорционально длинам обрабатываемых строк.

В связи с этим, при недостаточном объеме памяти, рекомендуются использовать рекурсивные алгоритмы, так как они не используют дополнительной памяти в процессе работы.

Кроме того, итеративный алгоритм Дамерау – Левенштейна расходует больше памяти, что связано с хранением третьей строки "кэша"для учета допролнительной операции.

4.5 Сравнительный анализ алгоритмов

Приведенные характеристики показывают нам, что рекурсивная реализация алгоритма очень сильно проигрывает по времени. В связи с этим, рекурсивные алгоритмы следует использовать лишь для малых размерностей строк (1-2 символа) или при малом объеме оперативной памяти.

Так как во время печати очень часто возникают ошибки связанные с транспозицией букв, алгоритм поиска расстояния Дамерау – Левенштейна является наиболее предпочтительным, не смотря на то, что он проигрывает по времени и памяти алгоритму Левенштейна.

По аналогии с первым абзацем можно сделать вывод о том, что рекуррентный алгоритм поиска расстояния Дамерау-Левенштейна будет более затратным по времени по сравнению с итеративной реализацией алгоритма поиска расстояния Дамерау –Левенштейна с кешированием.

4.6 Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти вышеизложенных алгоритмов. Наименее затратным по времени оказался рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна.

Для обработок малых длин строк (1 – 2 символа) предпочтительнее использовать рекурсивные алгоритмы, в то время как для остальных случаев рекомендуются использовать итеративные реализации. Однако, стоит учитывать дополнительные затраты по памяти, возникающие при использовании итеративных алгоритмов.

Заключение

Алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна являются самыми популярными алгоритмами, которые помогают найти редакторское расстояние.

В результате выполнения данной лабораторной работы были изучены алгоритмы поиска расстояний Левенштейна (Формула 1.1) и Дамерау – Левенштейна (Формула 1.3), построены схемы (Рисунок 2.2, Рисунок 2.4), соответствующие данным алгоритмам, также разобраны рекурсивные алгоритмы (Рисунок 2.1, Рисунок 2.3). Реализован программный продукт, который вычисляет дистанцию 4 способами.

В рамках выполнения работы решены следующие задачи.

- Изучены расстояния Левенштейна и Дамерау–Левенштейна.
- Реализованы алгоритмы поиска расстояний:
  - Рекурсивный алгоритм нахождения расстояния Левенштейна.
  - Итеративный алгоритм нахождения расстояния Левенштейна с кэшированием.
  - Рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна.
  - Итеративный алгоритм нахождения расстояния Дамерау – Левенштейна с кэшированием.
- Замерено процессорное время работы реализаций алгоритмов поиска расстояний.
- Проведено сравнение временных характеристик, а также затраченной памяти.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Левенштейн В. И.* Двоичные коды с исправлением выпадений, вставок и замещений символов //. Т. 163. — – М.: Доклады АН СССР, 1965. — Гл. 4. С. 845—848.
2. The Go Programming Language [Электронный ресурс]. — Режим доступа: `https://golang.org/` (дата обращения: 14.09.2021).
3. testing – The Go Programming Language [Электронный ресурс]. — Режим доступа: `https://golang.org/pkg/testing/` (дата обращения: 14.09.2021).
4. Процессор Intel® Core™ i5-8250U [Электронный ресурс]. — Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/124967/intel-core-i5-8250u-processor-6m-cache-up-to-3-40-ghz.html` (дата обращения: 14.09.2021).
5. Ubuntu: Enterprise Open Source and Linux [Электронный ресурс]. — Режим доступа: `https://ubuntu.com/` (дата обращения: 14.09.2021).
6. LINUX.ORG.RU – Русская информация об ОС Linux [Электронный ресурс]. — Режим доступа: `https://www.linux.org.ru/` (дата обращения: 14.09.2021).