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, искусственный интеллект и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по лабораторной работе № 1

по курсу «Анализ Алгоитмов»

на тему: «Расстояние Левенштейна и Дамерау-Левенштейна»

Студент ИУ7-55Б _____ Романов С. К.
(Группа) (Подпись, дата) (И. О. Фамилия)

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

2022 г.

Оглавление

ВВЕДЕНИЕ

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

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- исправления ошибок в слове
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов, хромосом и белков

Цель лабораторной работы:

– Изучение и исследование особенностей задач динамического программирования.

Задачи лабораторной работы:

1. Изучить и реализовать алгоритмы Дамерау-Левенштейна.
   - Итеративный метод поиска Дамерау-Левенштейна;
   - Рекурсивный метод поиска Дамерау-Левенштейна;
   - Рекурсивный с кешированием метод поиска Дамерау-Левенштейна
2. Создать ПО, реализующее алгоритмы, указанные в варианте.
3. Провести анализ затрат работы программы по времени и по памяти, выяснить влияющие на них характеристики.
4. Создать отчёт, содержащий:
   - схемы выбранных алгоритмов;
   - обоснование выбора технических средств;

– результаты тестирования;

– Выводы.

Для достижения поставленных целей и задач необходимо:

1. Изучить теоретические основы алгоритмов Дамерау-Левенштейна.
2. Реализовать выше обозначенные алгоритмы.
3. Провести экспериментальное исследование.

В ходе работы будут затронуты следующие темы:

1. Динамическое программирование.
2. Алгоритмы поиска Дамерау-Левенштейна.
3. Оценка реализаций алгоритмов.

1 Аналитическая часть

Расстояние Левенштейна [1] между двумя строками — это минимальное количество операций вставки, удаления и замены, необходимых для превращения одной строки в другую.

Цены операций могут зависеть от вида операции (вставка (insert), удаление (delete), замена (replace) и/или от участвующих в ней символов, отражая разную вероятность разных ошибок при вводе текста, и т. п. В общем случае:

- $w(a, b)$ — цена замены символа a на символ b.
- $w(\lambda, b)$ — цена вставки символа b.
- $w(a, \lambda)$ — цена удаления символа a.

Для решения задачи о редакционном расстоянии необходимо найти последовательность замен, минимизирующую суммарную цену. Расстояние Левенштейна является частным случаем этой задачи при

- $w(a, a) = 0$.
- $w(a, b) = 1,\ a \neq b$.
- $w(\lambda, b) = 1$.
- $w(a, \lambda) = 1$.

1.1 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:

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

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 D 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:
   (a) Сумма цены преобразования строки a в b и цены проведения операции удаления, которая необходима для преобразования a' в a;

(b) Сумма цены преобразования строки a в b и цены проведения операции вставки, которая необходима для преобразования b' в b;

(c) Сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются разные символы;

(d) Цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной ценой преобразования будет минимальное значение приведенных вариантов.

1.2 Матричный алгоритм нахождения расстояния Левенштейна

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 $D(i, j)$.

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием матричного алгоритма. Суть данного метода заключается в параллельном заполнении матрицы при выполнении рекурсии. В случае, если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, результат нахождения расстояния заносится в матрицу. В случае, если обработанные ранее данные встречаются снова, для них расстояние не находится и алгоритм переходит к следующему шагу.

1.4 Расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна может быть найдено по формуле (1.3), которая задана как

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

Формула выводится по тем же соображениям, что и формула (1.1). Как и в случае с рекурсивным методом, прямое применение этой формулы неэффективно по времени исполнения, то аналогично методу из (1.3) производится добавление матрицы для хранения промежуточных значений рекурсивной формулы.

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 данной части будут рассмотрены схемы алгоритмов нахождения расстояния Дамерау - Левенштейна. На рисунках 2.1 - 2.4 представлены рассматриваемые алгоритмы.

2.1 Итеративный алгоритм

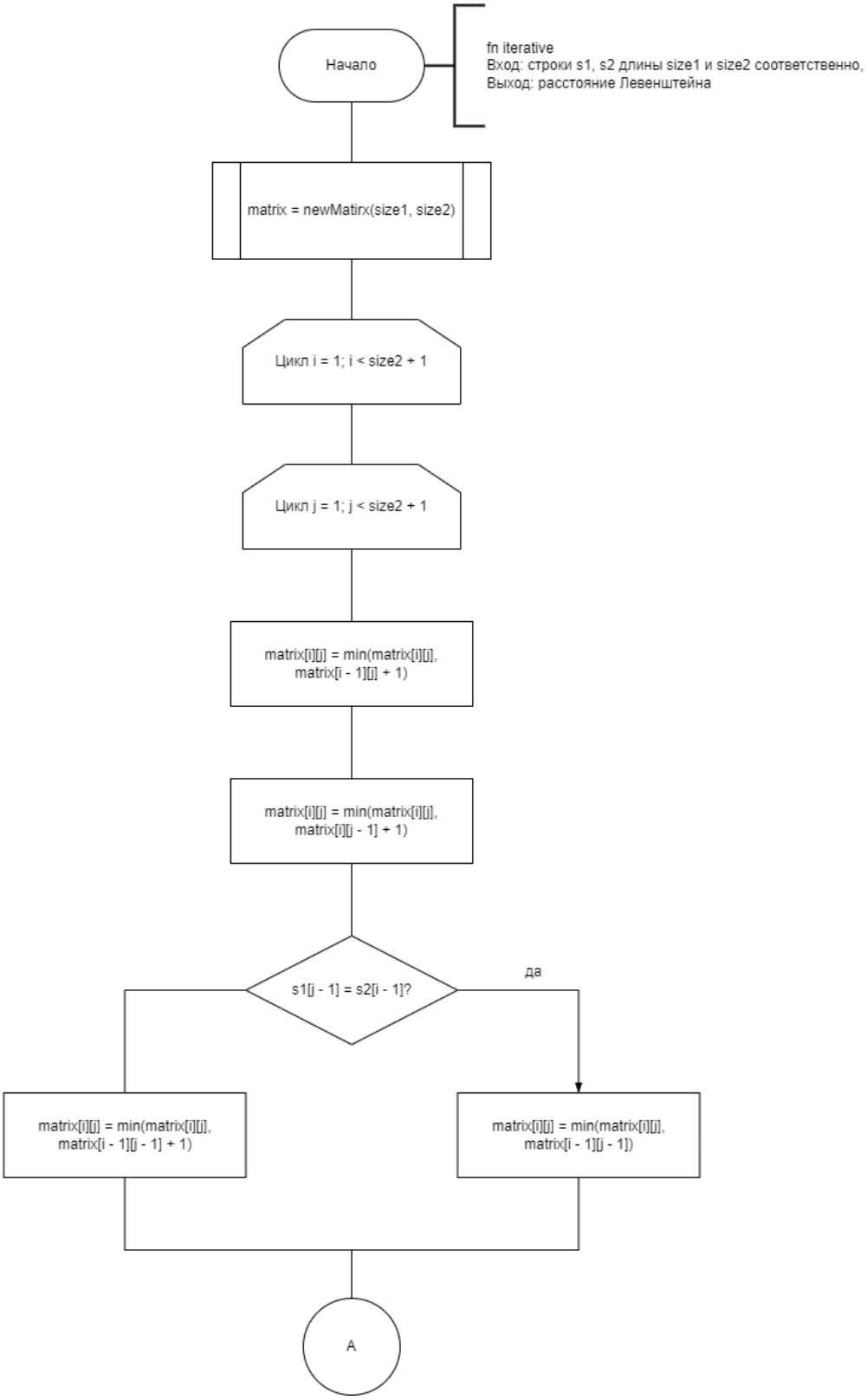


Рисунок 2.1 – Схема итеративного алгоритма нахождения расстояния Дамерау-Левенштейна, первая часть

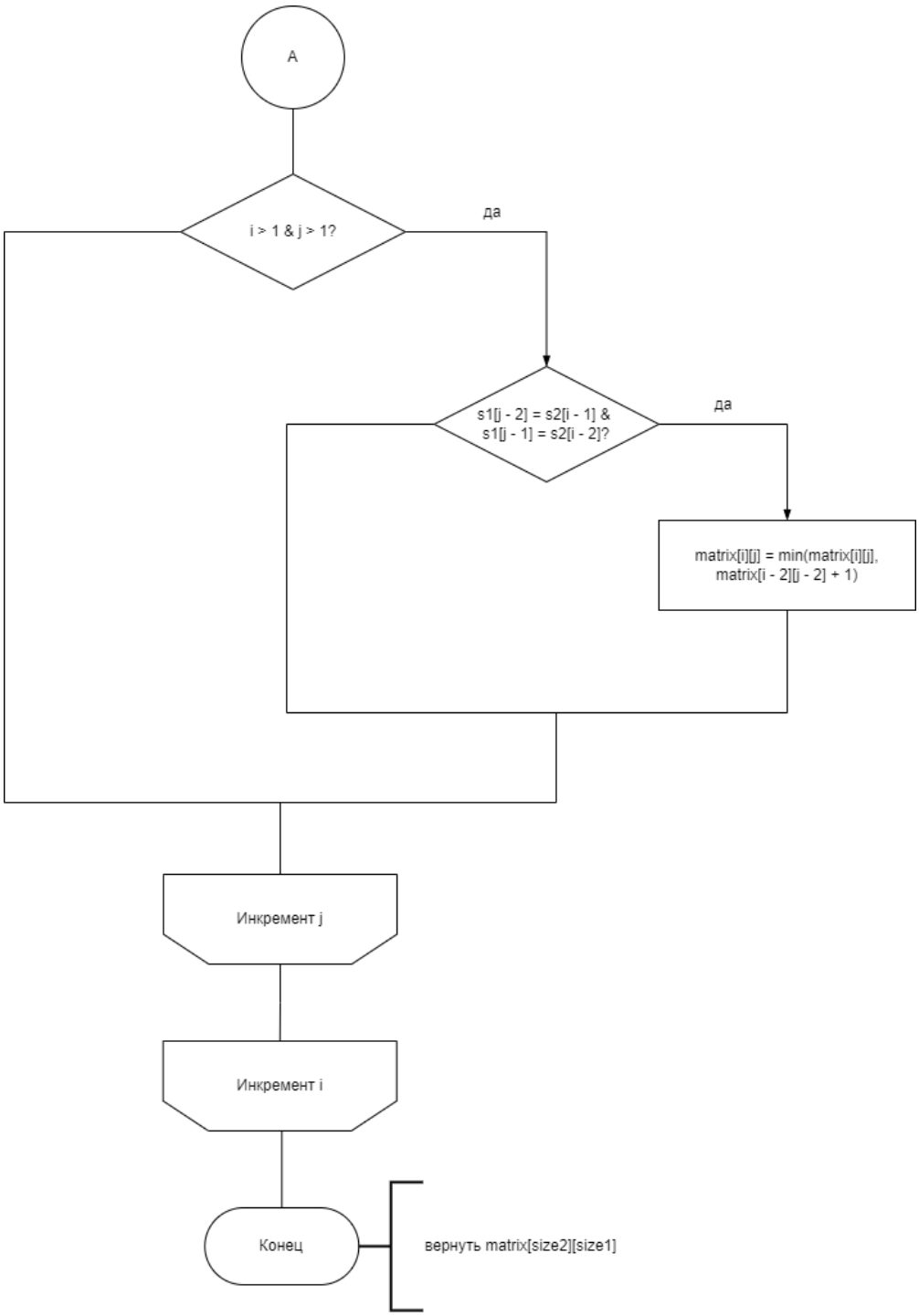


Рисунок 2.2 – Схема итеративного алгоритма нахождения расстояния Дамерау-Левенштейна, вторая часть

2.2 Рекурсивный алгоритм

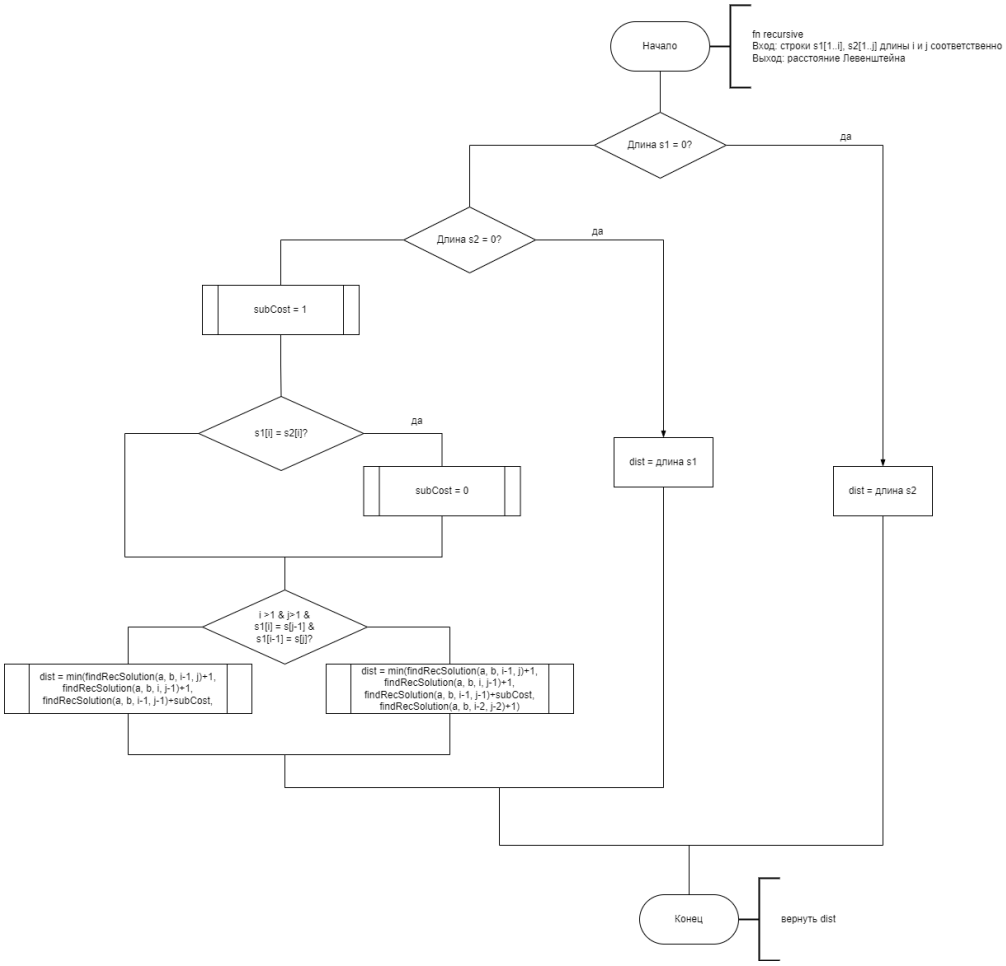


Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

2.3 Рекурсивный алгоритм с кэшированием

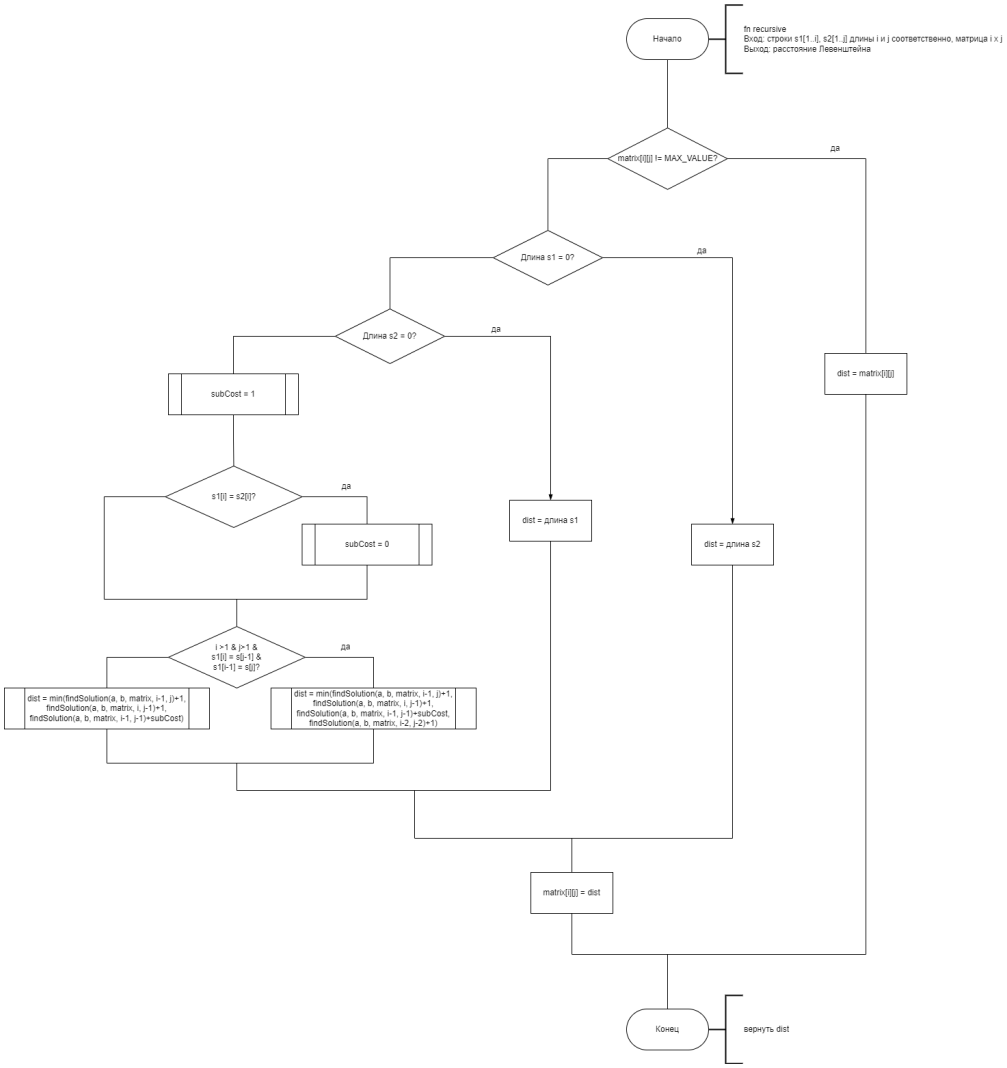


Рисунок 2.4 – Схема рекурсивного алгоритма с кэшем нахождения расстояния Дамерау-Левенштейна

Вывод

На основе теоретических данных, полученные в аналитическом разделе были построены схемы иследуеммых алгоритмов.

3 Технологическая часть

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

3.1 Требования к программному обеспечению

К программе предъявляется ряд условий:

– На вход подаются две строки в любой раскладке (в том числе и пустые);

– На выход ПО должно выводить полученное расстояние и вспомогательны матрицы;

3.2 Средства реализации

Для реализации данной лабораторной работы необходимо установить следующее программное обеспечение:

- Golang 1.19.1 - язык программирования
- Benchdraw 0.1.0 - Средство визуализации данных
- LaTeX - система документооборота

3.3 Листинги кода

В листингах 3.1, 3.2 и 3.3 приведены реализации алгоритмов сортировки

— "Итеративный метод поиска Дамерау-Левенштейна";

— "Рекурсивный метод поиска Дамерау-Левенштейна";

— "Рекурсивный с кешированием метод поиска Дамерау-Левенштейна";

соответственно

3.3.1 Итеративный метод поиска Дамерау-Левенштейна

Листинг 3.1 – Итеративный метод поиска Дамерау-Левенштейна

```
1
2  func DL_STD(a, b string) (int, Matrix) {
3      matrix := NewMatrix(len(a), len(b))
4
5      for i := 0; i < len(a)+1; i++ {
6          matrix.m[i][0] = i
7      }
8
9      for i := 1; i < len(b)+1; i++ {
10         matrix.m[0][i] = i
11     }
12
13     for i := 1; i < len(a)+1; i++ {
14         for j := 1; j < len(b)+1; j++ {
15             subCost := 1
16             if a[i-1] == b[j-1] {
17                 subCost = 0
18             }
19             matrix.m[i][j] = min(matrix.m[i-1][j]+1,
20                 matrix.m[i][j-1]+1,
21                 matrix.m[i-1][j-1]+subCost,
22             )
23             if i > 1 && j > 1 && a[i-1] == b[j-2] && a[i-2] == b[j-1] {
24                 matrix.m[i][j] = min(matrix.m[i][j], matrix.m[i-2][j-2]+1)
25             }
26         }
27     }
28
29     return matrix.m[len(a)][len(b)], *matrix
30 }
```

3.3.2 Рекурсивный метод поиска Дамерау-Левенштейна

Листинг 3.2 – Рекурсивный метод поиска Дамерау-Левенштейна

```
1 func findRecSolution(a, b string, i, j int) int {
2
3     if min(i, j) < 0 {
```

```
        return math.MaxInt16
    }

    if i == 0 {
        return j
    }

    if j == 0 {
        return i
    }

    subCost := 1
    if a[i-1] == b[j-1] {
        subCost = 0
    }

    if i > 1 && j > 1 && a[i-1] == b[j-2] && a[i-2] == b[j-1] {
        return min(findRecSolution(a, b, i-1, j)+1,
            findRecSolution(a, b, i, j-1)+1,
            findRecSolution(a, b, i-1, j-1)+subCost,
            findRecSolution(a, b, i-2, j-2)+1)
    }

    return min(findRecSolution(a, b, i-1, j)+1,
        findRecSolution(a, b, i, j-1)+1,
        findRecSolution(a, b, i-1, j-1)+subCost,
    )
}

func DL_Recursive(a, b string) int {
    return findRecSolution(a, b, len(a), len(b))
}
```

3.3.3 Рекурсивный с кешированием метод поиска Дамерау-Левенштейна

Листинг 3.3 – Рекурсивный с кешированием метод поиска Дамерау-Левенштейна

```
func findSolution(a, b string, mat *Matrix, i, j int) int {
    if mat.m[i][j] != math.MaxInt16 {
        return mat.m[i][j]
    }
    m := min(i, j)
    if m < 0 {
```

```
        return math.MaxInt16
    }
    if m == 0 {
        mat.m[i][j] = max(i, j)
        return mat.m[i][j]
    }

    subCost := 1
    if a[i-1] == b[j-1] {
        subCost = 0
    }

    if i > 1 && j > 1 && a[i-1] == b[j-2] && a[i-2] == b[j-1] {
        mat.m[i][j] = min(
            findSolution(a, b, mat, i-1, j)+1,
            findSolution(a, b, mat, i, j-1)+1,
            findSolution(a, b, mat, i-1, j-1)+subCost,
            findSolution(a, b, mat, i-2, j-2)+1)
        return mat.m[i][j]
    }

    mat.m[i][j] = min(findSolution(a, b, mat, i-1, j)+1,
        findSolution(a, b, mat, i, j-1)+1,
        findSolution(a, b, mat, i-1, j-1)+subCost,
    )

    return mat.m[i][j]
}

func DL_CacheRecursive(a, b string) (int, Matrix) {
    mat := NewMatrix(len(a), len(b))
    for i := 0; i < len(a)+1; i++ {
        for j := 0; j < len(b)+1; j++ {
            mat.m[i][j] = math.MaxInt16
        }
    }
    findSolution(a, b, mat, len(a), len(b))

    return mat.m[len(a)][len(b)], *mat
}
```

3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные, на которых было протестировано разработанное ПО.

Таблица 3.1 – Таблица тестовых данных

№	Первое слово	Второе слово	Ожидаемый результат	Полученный результат
1			0 0 0	0 0 0
2	kot	skat	2 2 2	2 2 2
3	kate	ktae	1 1 1	1 1 1
4	abacaba	aabcaab	2 2 2	2 2 2
5	sobaka	sboku	3 3 3	3 3 3
6	qwerty	queue	4 4 4	4 4 4
7	apple	aplpe	1 1 1	1 1 1
8		cat	3 3 3	3 3 3
9	parallels		9 9 9	9 9 9

Вывод

В данном разделе были разработаны исходные коды трех алгоритмов: вычисления расстояния Дамерау-Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы.

4 Исследовательская часть

Ниже приведены технические характеристики устройства, на котором было проведено тестирование ПО:

- Операционная система: Windows 11 используя Windows Subsystem for Linux 2 (WSL2) [2] Имитирующей Arch Linux [3] 64-bit.
- Оперативная память: 16 Гб.
- Процессор: 11th Gen Intel(R) Core(TM) i5-11320H @ 3.20 ГГц [4].

4.1 Время выполнения алгоритмов

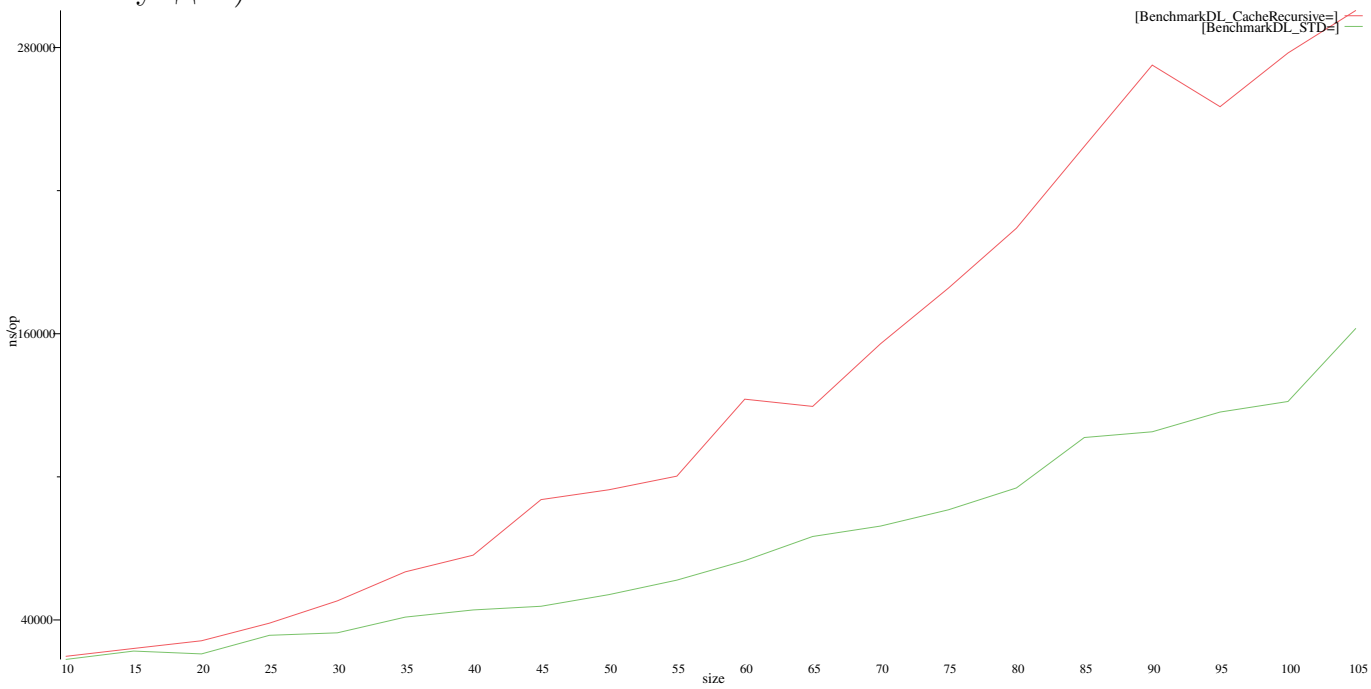
Алгоритмы тестировались при помощи ”бенчмарков” предоставляемых встренными средствами языка Go [5]. Пример такого "бенчмарка приведен в листинге 4.1. Количество повторов регулируется тестирующей системой самостоятельно, хотя при желание оные можно задать. Точное количество повторов определяется в зависимости от того, был ли получен стабильный результат согласно системе.

Листинг 4.1 – Пример бенчмарка

```
func BenchmarkDL_STD(b *testing.B) {
    for steps, amount := 0, BEGIN; steps < STEPS; steps++ {
        amount += INC
        b.Run(fmt.Sprintf("size=%d", amount), func(b *testing.B) {
            //sample_a := GenerateString(amount)
            //sample_b := GenerateString(amount)
            b.ResetTimer()
            for i := 0; i < b.N; i++ {
                sample_a := GenerateString(amount)
                sample_b := GenerateString(amount)

                DL_STD(sample_a, sample_b)
            }
        })
    }
}
```

График времени выполнения алгоритмов на случайных данных (в наносекундах)



4.2 Использование памяти

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Поэтому, максимальный расход памяти равен:

$$(\mathcal{S}(STR_1) + \mathcal{S}(STR_2)) \cdot (2 \cdot \mathcal{S}(\text{string}) + 3 \cdot \mathcal{S}(\text{integer})), \tag{4.1}$$

где $\mathcal{S}$ — оператор вычисления размера, STR_1, STR_2 — строки, string — строковый тип,
integer — целочисленный тип.

Использование памяти при итеративной реализации теоретически равно:

$$(\mathcal{S}(STR_1) + 1) \cdot (\mathcal{S}(STR_2) + 1) \cdot \mathcal{S}(\text{integer}) + 5 \cdot \mathcal{S}(\text{integer}) + 2 \cdot \mathcal{S}(\text{string}). \tag{4.2}$$

Вывод

Рекурсивная реализация алгоритма нахождения расстояния Левенштейна работает дольше остальных реализаций, а время работы этой реализации увеличивается в геометрической прогрессии. Это происходит из-за того, что

при рекурсивной реализации алгоритма на каждом шаге необходимо заново вычислять расстояние Левенштейна для подстрок, что приводит к большому количеству вызовов функции.

Также стоит отметить, что рекурсивная реализация алгоритма нахождения расстояния Левенштейна не подходит для больших строк, так как при больших значениях глубины стека вызовов возникает переполнение стека.

Согласно графикам, время выполнения рекурсивного алгоритма с кешем также работает медленнее итеративного. Связано это с тем, что в рекусривной версии алгоритма присутствуют дополнительные затраты памяти и процессорного времени на вызовы функций.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были достигнуты следующие задачи:

- были теоретически изучены алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;
- для некоторых реализаций были применены методы динамического программирования, что позволило сделать алгоритмы быстрее;
- на основе полученных в ходе экспериментов данных были сделаны выводы по поводу эффективности всех реализованных алгоритмов;
- был подготовлен отчет по Лабораторной работе.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Левенштейн В. И.* Двоичные коды с исправлением выпадений, вставок и замещений символов //. Т. 163. — Доклады АН СССР, 1965. — Гл. 4. С. 845—848.
2. Windows Subsystem for Linux 2 [Электронный ресурс]. — Дата обращения: 19.09.2022. Режим доступа: `https://learn.microsoft.com/en-us/windows/wsl/about#what-is-wsl-2`.
3. Arch Linux [Электронный ресурс]. — Дата обращения: 19.09.2022. Режим доступа: `https://archlinux.org/`.
4. Процессор Intel® Core™ i5-11320H [Электронный ресурс]. — Дата обращения: 19.09.2022. Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/217183/intel-core-i511320h-processor-8m-cache-up-to-4-50-ghz-with-ipu.html`.
5. Go [Электронный ресурс]. — Дата обращения: 19.09.2022. Режим доступа: `https://go.dev/`.