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)

ОТЧЁТ

По лабораторной работе №1

По курсу: «Анализ алгоритмов»

Тема: «Расстояние Левенштейна»

Студент: Керимов А. Ш.

Группа: ИУ7-54Б

Преподаватели: Волкова Л. Л.,
Строганов Ю. В.

Москва, 2019

Оглавление

Введение

Расстояние Левенштейна между двумя строками в теории информации и компьютерной лингвистике — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

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

Расстояние Левенштейна и его обобщения активно применяются: для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи); для сравнения текстовых файлов утилитой `diff` и ей подобными (здесь роль «символов» играют строки, а роль «строк» — файлы); в биоинформатике для сравнения генов, хромосом и белков [2].

Расстояние Дамерау — Левенштейна является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавлена операция транспозиции (перестановки) символов. Дамерау показал, что 80% человеческих ошибок при наборе текстов составляют перестановки соседних символов, пропуск символа, добавление нового символа, и ошибка в символе.

Задачи работы

- Изучение алгоритмов Левенштейна и Дамерау–Левенштейна.
- Применение методов динамического программирования для реализации алгоритмов.
- Получение практических навыков реализации алгоритмов Левенштейна и Дамерау — Левенштейна.
- Сравнительный анализ алгоритмов на основе экспериментальных данных.
- Подготовка отчета по лабораторной работе.

1 Аналитическая часть

Как говорилось ранее, расстояние Левенштейна 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, удаление, замена) 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

Расстояние Левенштейна является частным случаем этой задачи [3] 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 $D(|a|, |b|)$, где $|a|$ означает длину строки a; $a[i]$ — i-ый

символ строки a , функция $D(i,j)$ определена как:

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

а функция $m(a,b)$ определена как

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

Рекурсивный алгоритм реализует данную формулу. 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, аналогично, для перевода из строки a в пустую требуется $|a|$ операций;
3. для перевода из строки a в строку b требуется выполнить последовательно некоторое кол-во операций (удаление, вставка, замена) в некоторой последовательности. Как можно показать сравнением, последовательность проведения любых двух операций можно поменять, и, как следствие, порядок проведения операций не имеет никакого значения. Тогда цена преобразования из строки a в строку b может быть выражена как (полагая, что a', b' — строки a и b без последнего символа соответственно):
   - 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 b', предполагая, что a и b оканчиваются на один и тот же символ.

Очевидно, что минимальной ценой преобразования будет минимальное значение этих вариантов.

1.2 Алгоритм Вагнера — Фишера (построчный)

Прямая реализация приведенной выше формулы D может быть малоэффективна 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 $D(i, j)$.

Можно заметить, что при каждом заполнении новой строки значения предыдущей становятся ненужными. Поэтому можно провести оптимизацию по памяти и использовать дополнительно только одномерный массив размером $|b|$. Такой вариант алгоритма называется построчным и именно он реализован в данной работе в качестве нерекурсивного.

1.3 Расстояния Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна может быть найдено по рекурсивной формуле $d_{a,b}(|a|,|b|)$, где $d_{a,b}(i,j)$ задана как

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

где $M_{a,b}(i,j)$ задается как

$$M_{a,b}(i,j) = \begin{cases} d_{a,b}(i-2,j-2)+1 & \text{если } i,j>1; a[i]=b[j-1]; b[j]=a[i-1], \\ +\infty & \text{иначе} \end{cases}. \tag{1.4}$$

Формула выводится по тем же соображениям, что и формула (1.1). Т.к. прямое применение этой формулы неэффективно, то аналогично действиям из предыдущего пункта производится добавление матрицы для хранения промежуточных значений рекурсивной формулы и оптимизация по памяти. В таком случае необходимо хранить одномерный массив длиной $3\min(|a|,|b|)$.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна $d(S_1,S_2)$ можно подсчитать по рекуррентной формуле $d(S_1,S_2) = D(M,N)$, где

$$D(i,j) = \begin{cases} 0, & i = 0,\ j = 0 \\ i, & i > 0,\ j = 0 \\ j, & i = 0,\ j > 0 \\ \min\{ & \\ \quad D(i, j-1) + 1, & \\ \quad D(i-1, j) + 1, & i > 0,\ j > 0 \\ \quad D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ \} & \end{cases} \tag{1.5}$$

Вывод

Формулы Левенштейна и Дамерау — Левенштейна для рассчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

2.1 Схема алгоритма Левенштейна

На рисунке 2.2 приведена схема матричного алгоритма Левенштейна.

2.2 Схема алгоритма Дамерау — Левенштейна

На рисунках 2.6 и 2.5 представлены матричный и рекурсивный алгоритмы Дамерау — Левенштейна.

Вывод

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

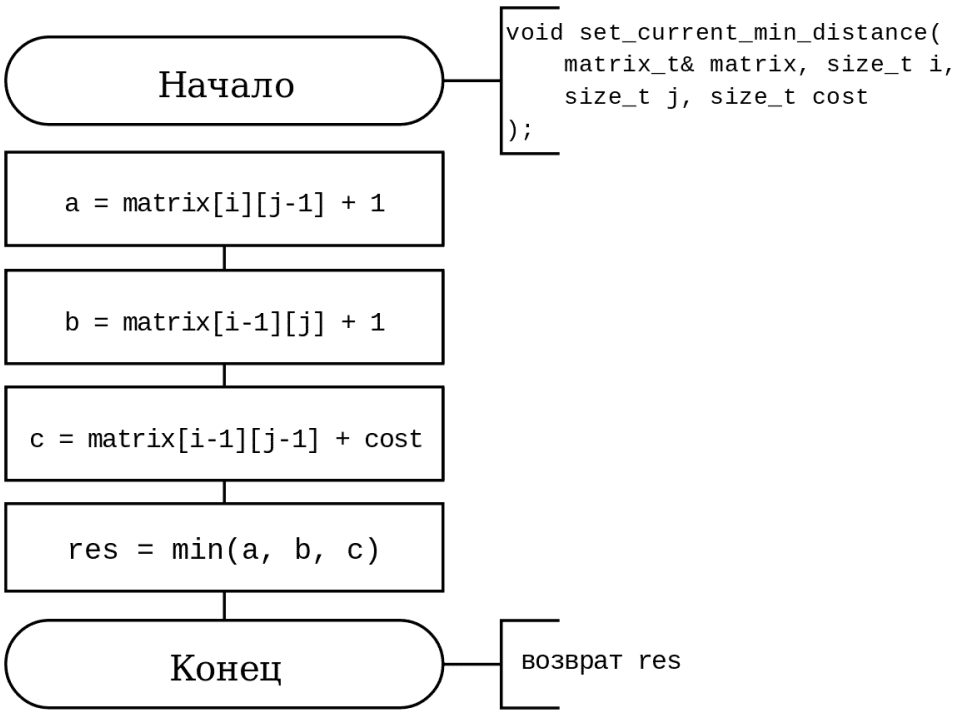

Рис. 2.1: Схема процедуры, вычисляющей текущее расстояние

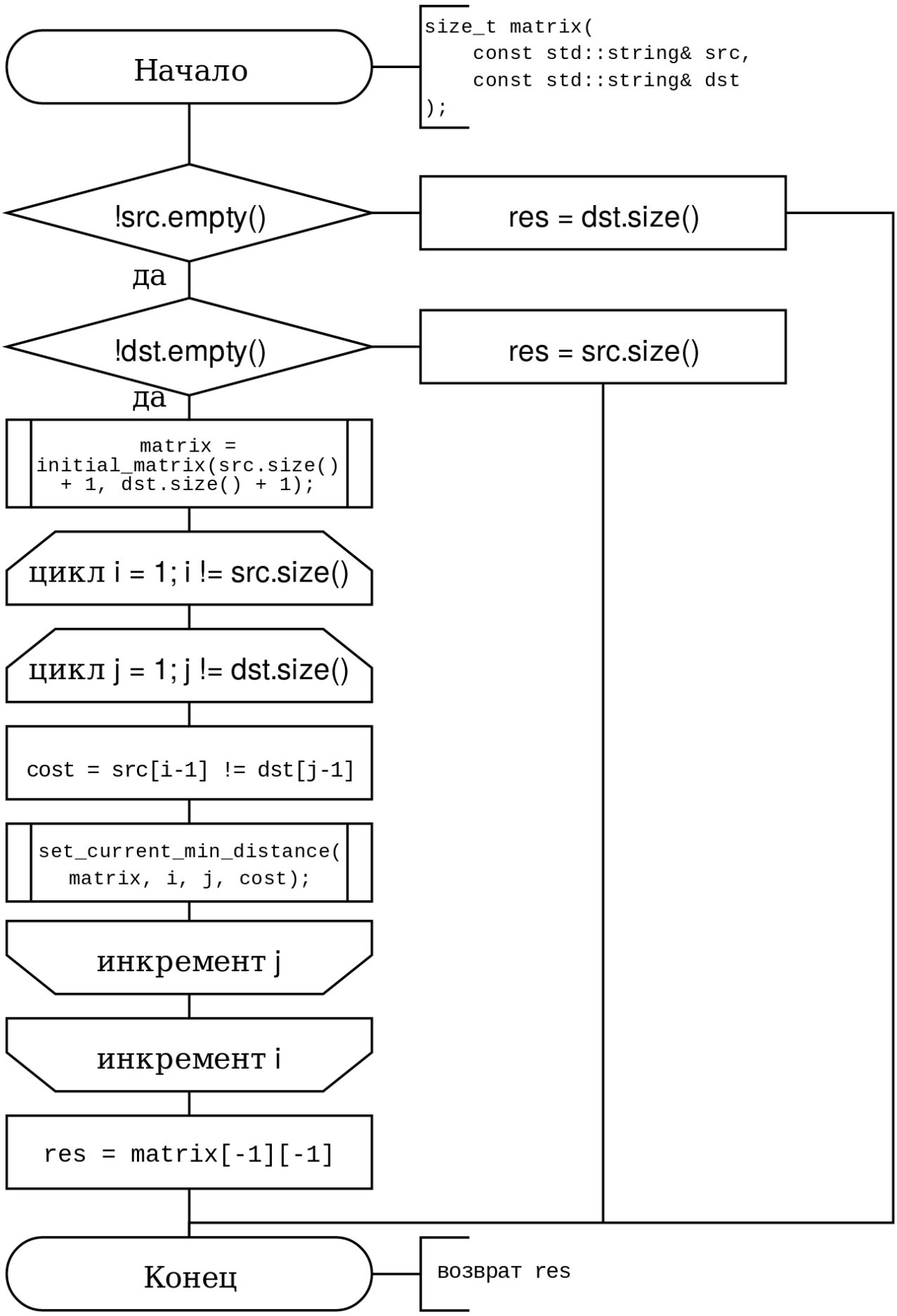

Рис. 2.2: Схема матричного алгоритма Левенштейна

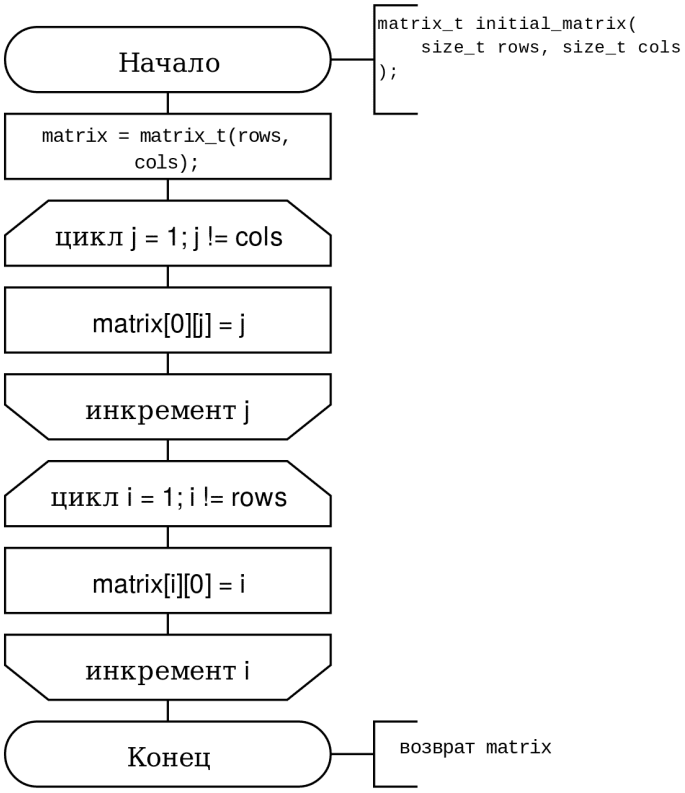

Рис. 2.3: Схема функции, создающей матрицу с инициализированными первой строкой и первым столбцом

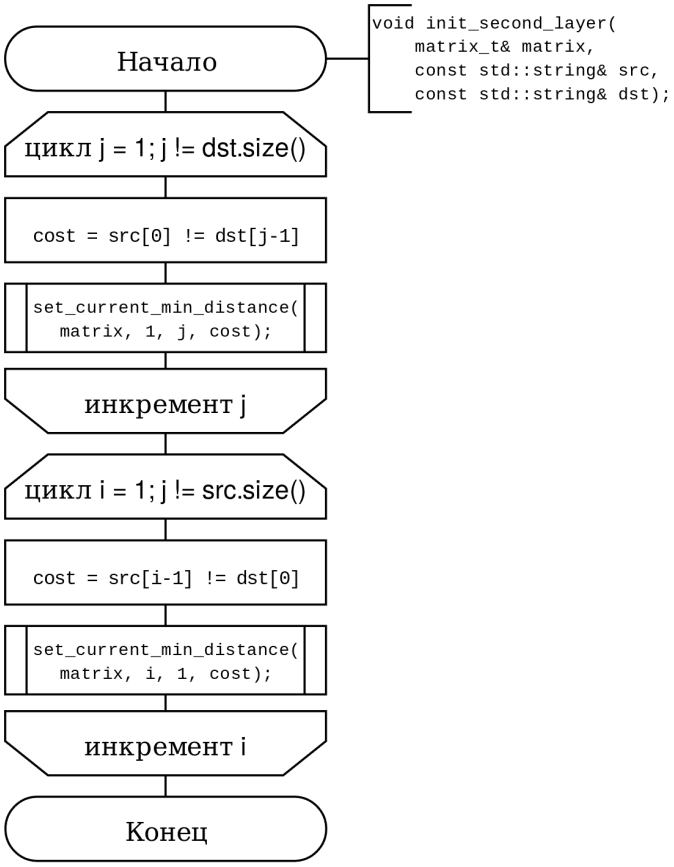

Рис. 2.4: Схема процедуры, инициализирующей вторую строку и второй столбец

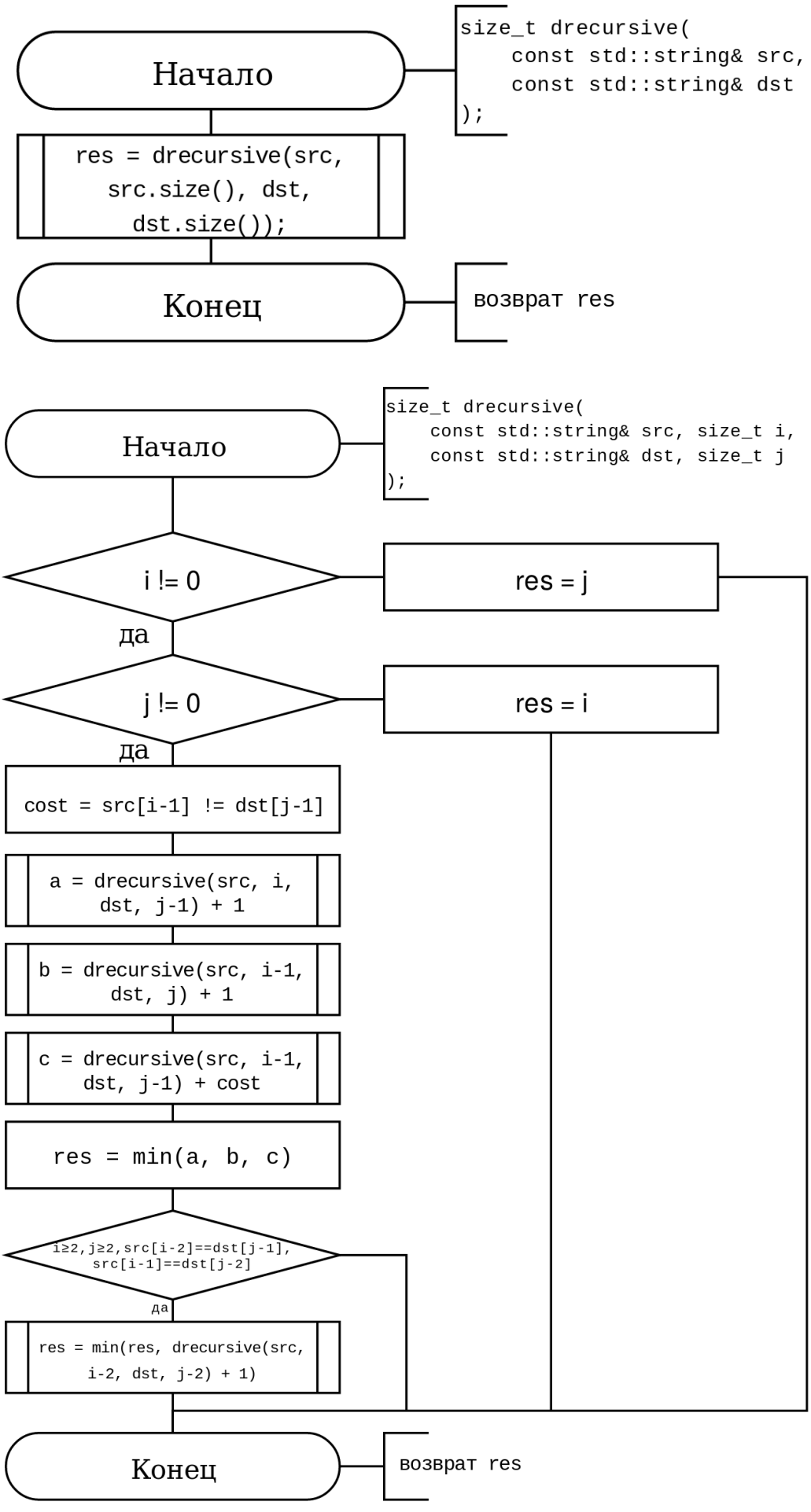

Рис. 2.5: Схема рекурсивного алгоритма Дамерау — Левенштейна

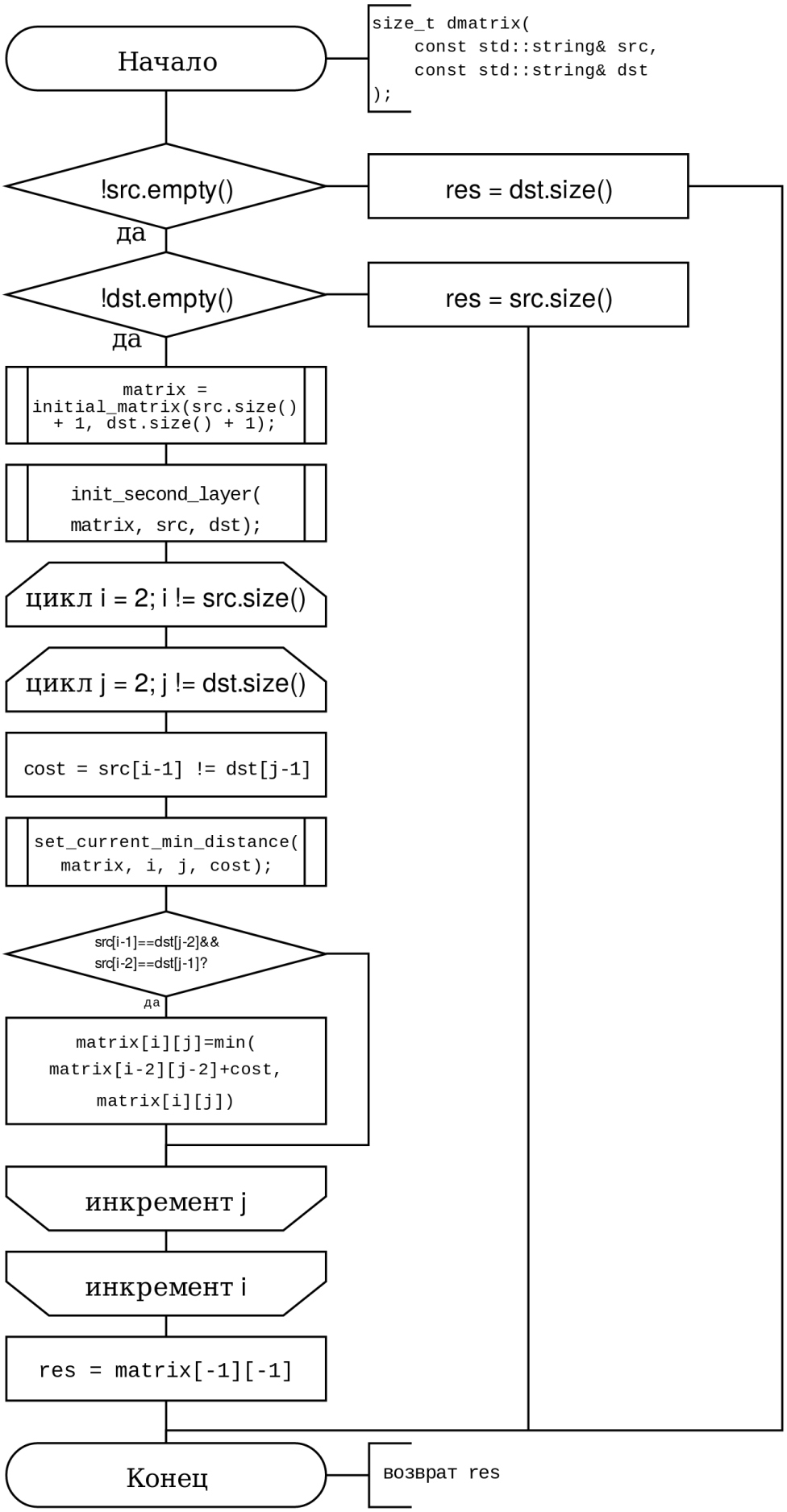

Рис. 2.6: Схема матричного алгоритма Дамерау — Левенштейна

3 Технологическая часть

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

3.1 Требования к ПО

К программе предъявляется ряд требований:

- на вход подаётся тип алгоритма (`[damerau_]<matrix|recursive>`) и две строки;
- на выходе — искомое расстояние (для `matrix` — Левенштейна , для `damerau_matrix` или `damerau_recursive` — Дамерау — Левенштейна);
- при указании ключа `-v|--verbose` в матричных алгоритмах необходимо вывести матрицу;
- программа не должна аварийно завершаться при некорректном вводе пользователя.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран высокопроизводительный язык C++ [4], предоставляющий широкие возможности для эффективной реализации алгоритмов.

3.3 Листинг кода

В листингах 3.1–3.4 приведены реализации алгоритмов Левенштейна и Дамерау — Левенштейна.

```
namespace levenshtein::core {
inline namespace detail {
```

```
matrix_t matrix_with_initialized_first_layer(size_t rows, size_t cols) {
    matrix_t matrix(rows, row_t(cols));
    for (size_t j = 1; j < matrix[0].size(); ++j) {
        matrix[0][j] = j;
    }
    for (size_t i = 1; i < matrix.size(); ++i) {
        matrix[i][0] = i;
    }
    return matrix;
}

void initialize_second_layer(matrix_t& matrix, const std::string& src, const
    std::string& dst) {
    for (size_t j = 1; j <= dst.size(); ++j) {
        set_current_min_distance(matrix, 1, j, src[0] != dst[j - 1]);
    }
    for (size_t i = 1; i <= src.size(); ++i) {
        set_current_min_distance(matrix, i, 1, src[i - 1] != dst[0]);
    }
}

matrix_t initial_matrix(size_t rows, size_t cols) {
    matrix_t matrix(rows, row_t(cols));

    return matrix;
}

void set_current_min_distance(matrix_t& matrix, size_t i, size_t j, size_t cost) {
    matrix[i][j] = std::min({
        matrix[i][j - 1] + 1,
        matrix[i - 1][j] + 1,
        matrix[i - 1][j - 1] + cost
    });
}

} // namespace detail
} // namespace levenshtein::core
```

Листинг 3.1: Вспомогательные функции для матричных алгоритмов

```
namespace levenshtein::core {

matrix_t matrix(const std::string& src, const std::string& dst) {
    auto matrix = matrix_with_initialized_first_layer(src.size() + 1, dst.size() + 1);

    if (src.empty() || dst.empty()) {
        return matrix;
    }
```

```
    for (size_t i = 1; i <= src.size(); ++i) {
        for (size_t j = 1; j <= dst.size(); ++j) {
            set_current_min_distance(matrix, i, j, src[i - 1] != dst[j - 1]);
        }
    }

    return matrix;
}

} // namespace levenshtein::core
```

Листинг 3.2: Функция, реализующая матричный алгоритм Левенштейна

```
namespace levenshtein::core {

matrix_t damerau_matrix(const std::string& src, const std::string& dst) {
    auto matrix = matrix_with_initialized_first_layer(src.size() + 1, dst.size() + 1);

    if (src.empty() || dst.empty()) {
        return matrix;
    }

    initialize_second_layer(matrix, src, dst);

    for (size_t i = 2; i <= src.size(); ++i) {
        for (size_t j = 2; j <= dst.size(); ++j) {
            const size_t cost = src[i - 1] != dst[j - 1];
            set_current_min_distance(matrix, i, j, cost);

            if (src[i - 1] == dst[j - 2] && src[i - 2] == dst[j - 1]) {
                matrix[i][j] = std::min(matrix[i - 2][j - 2] + cost, matrix[i][j]);
            }
        }
    }

    return matrix;
}

} // namespace levenshtein::core
```

Листинг 3.3: Функция, реализующая матричный алгоритм Дамерау — Левенштейна

```
namespace levenshtein::core {
inline namespace detail {

size_t damerau_recursive(const std::string& src, size_t i, const std::string& dst,
    size_t j) {
    if (!i || !j) return i + j;
```

```
    size_t dist = std::min({
        damerau_recursive(src, i - 1, dst, j) + 1,
        damerau_recursive(src, i, dst, j - 1) + 1,
        damerau_recursive(src, i - 1, dst, j - 1) + (src[i - 1] != dst[j - 1])
    });

    if (i >= 2 && j >= 2 && src[i - 2] == dst[j - 1] && src[i - 1] == dst[j - 2]) {
        dist = std::min(damerau_recursive(src, i - 2, dst, j - 2) + 1, dist);
    }

    return dist;
}

} // namespace detail

size_t damerau_recursive(const std::string& src, const std::string& dst) {
    return damerau_recursive(src, src.size(), dst, dst.size());
}

} // namespace levenshtein::core
```

Листинг 3.4: Функция, реализующая рекурсивный алгоритм Дамерау — Левенштейна

В листинге 3.5 приведена реализация функции замера времени работы алгоритмов

```
template <typename F, typename... Args>
double time(size_t N, F func, Args&&... args) {
    if (!N) {
        return 0;
    }

    const auto start = std::chrono::high_resolution_clock::now();

    for (size_t i = 0; i != N; ++i) {
        func(std::forward<Args>(args)...);
    }

    const auto end = std::chrono::high_resolution_clock::now();

    return static_cast<double>(std::chrono::duration_cast<std::chrono::nanoseconds>(end
        - start).count()) / 1.0e6 / N;
}
```

Листинг 3.5: Функция замера времени работы алгоритмов в тиках

Строка 1	Строка 2	Ожидаемый результат	
		Левенштейн	Дамерау — Левенштейн
cook	cooker	2	2
mother	money	3	3
woman	water	4	4
program	friend	6	6
house	girl	5	5
probelm	problem	2	1
head	ehda	3	2
bring	brought	4	4
happy	happy	0	0
minute	moment	5	5
person	eye	5	5
week	weeks	1	1
member	morning	6	6
death	health	2	2
education	question	4	4
room	moor	2	2
car	city	3	3
air	area	3	3

Таблица 3.1: Функциональные тесты

В таблице 3.1 приведен функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дамерау — Левенштейна. Все тесты пройдены успешно.

Вывод

Были разработаны и протестированы схемы трех алгоритмов: вычисления расстояния Левенштейна матрицей, вычисления расстояния Дамерау — Левенштейна матрицей и рекурсивно.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунках 4.1—4.3.

```
/home/user/bmstu/AA/lab01/cmake-build-debug/levenshtein -v matrix russia great
                g     r     e     a     t
          0     1     2     3     4     5
     r    1     1     1     2     3     4
     u    2     2     2     2     3     4
     s    3     3     3     3     3     4
     s    4     4     4     4     4     4
     i    5     5     5     5     5     5
     a    6     6     6     6     5     6
Levenshtein distance: 6

Process finished with exit code 0
```

Рис. 4.1: Демонстрация работы итеративного алгоритма Левенштейна

```
/home/user/bmstu/AA/lab01/cmake-build-debug/levenshtein recursive russia great
6

Process finished with exit code 0
```

Рис. 4.2: Демонстрация работы рекурсивного алгоритма Левенштейна

```
/home/user/bmstu/AA/lab01/cmake-build-debug/levenshtein -v damerau russia great
                g     r     e     a     t
          0     1     2     3     4     5
     r    1     1     1     2     3     4
     u    2     2     2     2     3     4
     s    3     3     3     3     3     4
     s    4     4     4     4     4     4
     i    5     5     5     5     5     5
     a    6     6     6     6     5     6
Levenshtein distance: 6

Process finished with exit code 0
```

Рис. 4.3: Демонстрация работы алгоритма Дамерау — Левенштейна

4.2 Технические характеристики

- Операционная система: Ubuntu 19.10 64-bit.
- Память: 3,8 GiB.
- Процессор: Intel® Core™ i3-6006U CPU @ 2.00GHz

4.3 Время выполнения алгоритмов

Алгоритмы тестировались с помощью функции замера процессорного времени `std::chrono::high_resolution_clock` при количестве повторов 100 для `matrix`, `damerau_matrix`, и `damerau_recursive`;

Результаты замеров приведены в таблицах 4.1 и 4.2. На рисунках 4.4 и 4.5 приведены графики зависимостей времени работы алгоритмов от длины строк.

	Время, мс		
Длина строк	**matrix**	**damerau_matrix**	**damerau_recursive**
2	0.00280376	0.00293882	0.001849
3	0.00404252	0.00427016	0.00486
4	0.00581271	0.00592851	0.019472
5	0.00728198	0.00783907	0.093345
6	0.0117668	0.0133925	0.501891
7	0.0145187	0.0150317	2.56465
8	0.0150719	0.0154015	14.1558
9	0.0175882	0.0184566	100.05
10	0.0472756	0.0233616	461.319

Таблица 4.1: Замер времени для строк, размером до 10

4.4 Использование памяти

Алгоритмы Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, до-

Длина строк	Время, мс	
	matrix	**damerau_matrix**
10	0.0472756	0.0233616
20	0.0478761	0.0763991
30	0.106974	0.0991381
40	0.159155	0.170072
50	0.24294	0.351331
60	0.436508	0.392301
70	0.614052	0.498952
80	0.600738	0.701186
90	0.776267	1.07136
100	1.3964	1.42846
200	5.74174	6.92012
300	13.2496	13.7461
400	23.107	22.9783
500	26.5478	30.5103
600	33.5727	36.0928
700	45.7702	49.2277
800	59.5794	64.1255
900	75.3062	81.1656
1000	93.3559	100.582

Таблица 4.2: Замер времени для строк, размером до 1000

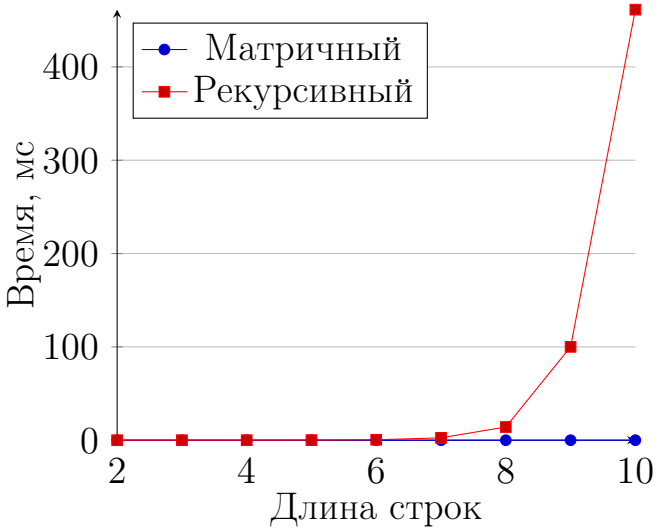

Рис. 4.4: Зависимость времени работы алгоритма вычисления расстояния Дамерау — Левенштейна от длины строк (матричная и рекурсивная реализации)

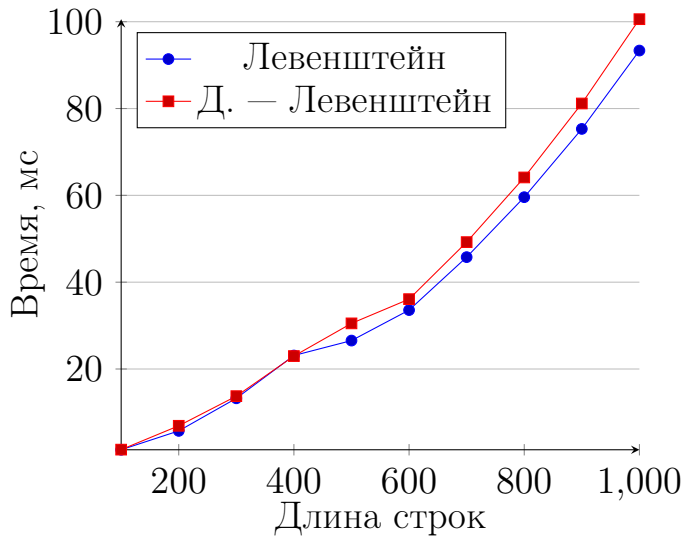

Рис. 4.5: Зависимость времени работы матричных реализаций алгоритмов Левенштейна и Дамерау — Левенштейна

статочно рассмотреть лишь разницу рекурсивной и матричной реализаций этих алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк, соответственно, максимальный расход памяти (4.1)

$$(\mathcal{S}(src) + \mathcal{S}(dst)) \cdot (2 \cdot \mathcal{S}(\text{std}::\text{string\&}) + 4 \cdot \mathcal{S}(\text{size\_t})), \tag{4.1}$$

где $\mathcal{S}$ — оператор вычисления размера, src, dst — строки, size_t — целочисленный тип, std::string& — тип указателя на строку.

Использование памяти при итеративной реализации теоретически равно

$$\mathcal{S}(src) \cdot \mathcal{S}(dst) \cdot \mathcal{S}(\text{size\_t}) + 2 \cdot (\mathcal{S}(\text{std}::\text{string\&}) + \mathcal{S}(\text{size\_t})). \tag{4.2}$$

Например, при вычислении расстояния Левеншейна для строк длиной 10, при размере типов size_t и std::string& 8 байтов получим:

- для рекурсивной реализации $(10 + 10) \cdot (2 \cdot 8 + 4 \cdot 8) = 960$ байтов;
- для матричной реализации $10 \cdot 10 \cdot 8 + 2 \cdot (8 + 8) = 832$ байта.

Можно заметить, что для корректной работы алгоритмов матрицу можно не хранить целиком, а использовать только текущую и предыдущую

строки. При этом, если менять местами src и dst в случае $\mathcal{S}(src) > \mathcal{S}(dst)$, то тогда расход памяти составит

$$(2\min(\mathcal{S}(src), \mathcal{S}(dst)) + 1) \cdot \mathcal{S}(\text{size\_t}) + 2 \cdot \mathcal{S}(\text{std::string\&}). \tag{4.3}$$

В примере выше для строк длиной 10: $(2(10 + 1) \cdot 8 + 2 \cdot 8) = 192$ байта.

Вывод

Рекурсивный алгоритм Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. Алгоритм Дамерау — Левенштейна работает немногим дольше алгоритма Левенштейна, т. к. в нём добавлены дополнительные проверки.

Но по расходу памяти итеративные алгоритмы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк. Однако в итеративных алгоритмах можно добиться ещё меньшего порядка роста, равному минимуму из длин строк, если не использовать всю матрицу целиком.

Заключение

В ходе выполнения работы были выполнены все поставленные задачи и изучены методы динамического программирования на основе алгоритмов вычисления расстояния Левенштейна.

Экспериментально были установлены различия в производительности различных алгоритмов вычисления расстояния Левенштейна. Рекурсивный алгоритм Левенштейна работает на несколько порядков медленнее матричной реализации. Если длина сравниваемых строк превышает 10, рекурсивный алгоритм становится неприемлимым для использования. Матричная реализация алгоритма Дамерау — Левенштейна работает дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки.

Теоретически было рассчитано использования памяти в каждом из алгоритмов вычисления расстояния Левенштейна. Обычные матричные алгоритмы потребляют намного больше памяти, чем рекурсивный, одна модфицированные их версии, держащие в памяти лишь текущую и предыдущую строки матрицы, и работают быстрее, и память используют меньше.

Литература

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] Гасфилд. Строки, деревья и последовательности в алгоритмах. Информатика и вычислительная биология. Невский Диалект БВХ-Петербург, 2003.

[3] Задача о редакционном расстоянии, алгоритм Вагнера-Фишера. Режим доступа: `shorturl.at/AD469`. Дата обращения: 15.10.2019.

[4] Working Draft, Standard for ProgrammingLanguage C++. Режим доступа: `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4713.pdf`. Дата обращения: 15.10.2019.