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 №1

по курсу «Анализы Алгоритмов»

на тему: «Редакционного растояния Левенштейна и Дамерау-Левенштейна»

Студент группы ИУ7-56Б _____ (Подпись, дата) Мамврийский И. С. (Фамилия И.О.)

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

Преподаватель _____ (Подпись, дата) Строганов Ю. В.. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

Расстояние Левенштейна – минимальное количество операций удаления, вставки одного символа и замены одного символа другим, необходимым для превращения одной строки в другую.

Оно применяется в теории информации и компьютерной лингвистике для решения следующих задач:

— исправление ошибок в слове(в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);

— сравнение текстовых файлов утилитой diff;

— для сравнения геномов, хромосом и белков в биоинформатике.

Целью данной лабораторной работы является описание и исследование алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Для достижения поставленнной цели необходимо выполнить следующие задачи:

1) Проанализировать алгоритмы поиска расстояний Дамерау-Левенштейна и Левенштейна;

2) Создать программное обеспечение, реализующее следующие алгоритмы:

   — нерекурсивный алгоритм поиска редакционного расстояния Левенштейна;

   — нерекурсивный алгоритм поиска редакционного расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм поиска редакционного расстояния Дамерау-Левенштейна;

   — рекурсивный с кэшированием алгоритм поиска редакционного расстояния Дамерау-Левенштейна.

3) Выбрать инструменты для замера процессорного времени выполнения реализаций алгоритмов;

4) Провести анализ затрат реализаций алгоритмов по времени и по памяти, определить влияющие на них характеристики.

1 Аналитическая часть

Расстояние Левенштейна(редакционное расстояние) – минимальное количество редакционных операций вставки, удаления и замены, необходимых для превращения одной строки в другую.[1]

Цены операций могут зависеть от вида операций и/или от участвующих в ней символов:

1) $w(a, b)$ — цена замены символа a на b;

2) $w(\lambda, b)$ — цена вставки символа b;

3) $w(a, \lambda)$ — цена удаления символа a.

Нам необходимо найти последовательность, минимализирующую суммарную цену. Расстояние Левенштейна является частным случаем решения данной зхадачи при:

— $w(a, a) = 0$;

— $w(a, b) = 1$, $a \neq b$, в противном случае замена не происходит;

— $w(\lambda, b) = 1$;

— $w(a, \lambda) = 1$.

Введем понятие M, которое будет обозначать совпадение, то есть $w(a, a) = 0$.

1.1 Матричный алгоритм нахождения расстояния Левенштейна

Расстояние между двумя строками a и b может быть вычислено по формуле 1.1 с использованием матрицы размером (N + 1) * (M + 1) для сохранения соответствующих промежуточных значений. Данный алгоритм представляет собой построчное заполнение матрицы $A_{|a||b|}$ значениями $D(i, j)$, где [i, j] – значение ячейки. Первая ячейчка заполняется 0, остальные в соответствии с формулой:

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

а функция 1.2 определена как:

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

Функция D составлена по следущему принципу, где a[i] – i-ый символ строки a, b[j] – j-ый символ строки b, функция D(i, j) определена как:

1) Из перевода пустой строки в пустую требуется 0 операций.

2) Из перевода пустой строки в строку a треубется $|a|$ операций.

3) Из перевода строки в пустую строку a треубется $|a|$ операций.

4) Для перевода из строки a в строку b требуется выполнить несколько операций(удаления, добавления, замены). Пусть a' и b' строки a и b без последнего символа, поэтому цена преобразования строки a в строку b может выглядеть следующшим образом:

   a) Сумма цены преобразования строки a в b и цена 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 цена операции замены, предполагая, что a и b оканчиваются на разные символы.

d) Сумма цены преобразоания из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

Минимальная цена преобразования – минимальное значение приведенных операций.

1.2 Рекурсивный алгоритм нахождения расстояния

Расстояние Дамерау-Левенштейна[1] может быть найдено по формуле 1.3, которая задана как:

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

Данная формула выводится по тем же соображенияем, что и формула 1.1, однако, в этой формуле добавляется еще одного условие для случая, когда обе строки не пустые.

Сумма цен преобразования из a" в b" и операции перестановки, предполагая, что длины a" и b" больше 1 и последние два символа a", если их поменять местами, совпадут с последними двумя символами b".

1.3 Рекурсивный алгоритм нахождения расстояния с кэшированием

Рекурсивная реализация алгоритма Домерау-Левенштейна малоэффективна при больших M и N, так как некоторые значения будут вычислены повторно. Для оптимизации данного алгоритма можно использовать кэш в виде матрицы для сохранения промежуточных значений. В таком случае алгоритм предстваляет собой рекурсивное заполнение матрицы $A_{|a||b|}$. промежуточными значениями D(i, j).

1.4 Нерекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Рекурсивная реализация алгоритма Левенштейна с кэшированием малоэффективна по времени при больших M и N. Для быстроты действия можно использовать нерекурсивный алгоритм. Он предстваляет собой итерационную реализацию заполнения матрицы промежуточными значениями D(i, j).

В качестве структуры данных для хранения можно использовать матрицу размером:

$$(N + 1) \times (M + 1), \tag{1.4}$$

Значение в ячейке [i, j] равно зачению $D(S1[1...i], S2[1...j])$. Первый элемент заполняется нулем, остальные в соответствии с формулой (1.3).

1.5 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который учитывает возможность перестановки соседних символов. Формулы Левенштейна и Дамерау-Левенштейна для рассчета расстояния задаются рекурсивно, следовательно, алгоритмы могут быть реализованы рекурсивно или итерационно.

2 Конструкторская часть

2.1 Алгоритмы нахождения редакционного расстояния

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

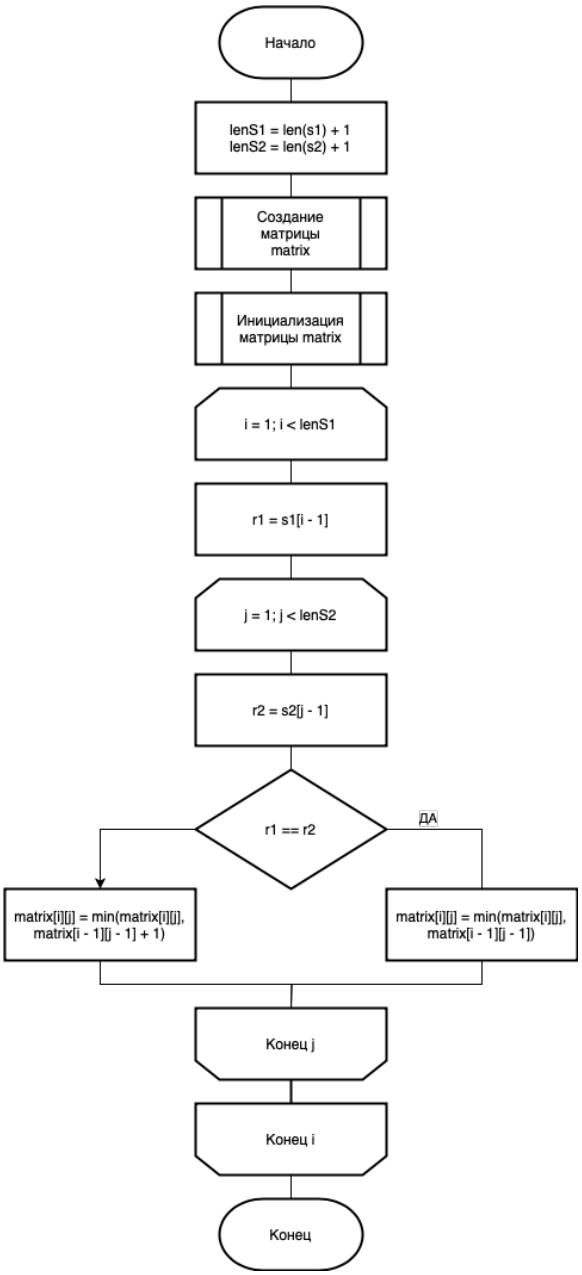

Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

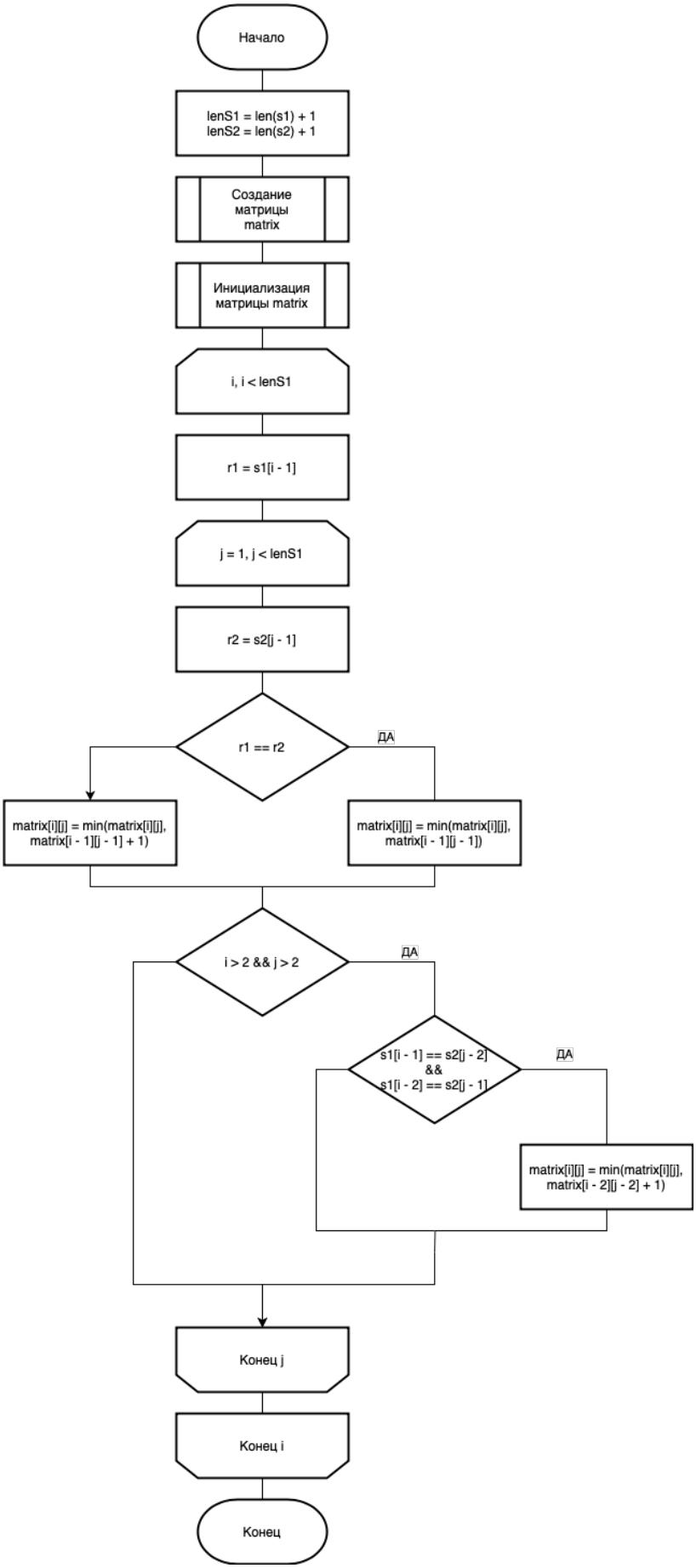

Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

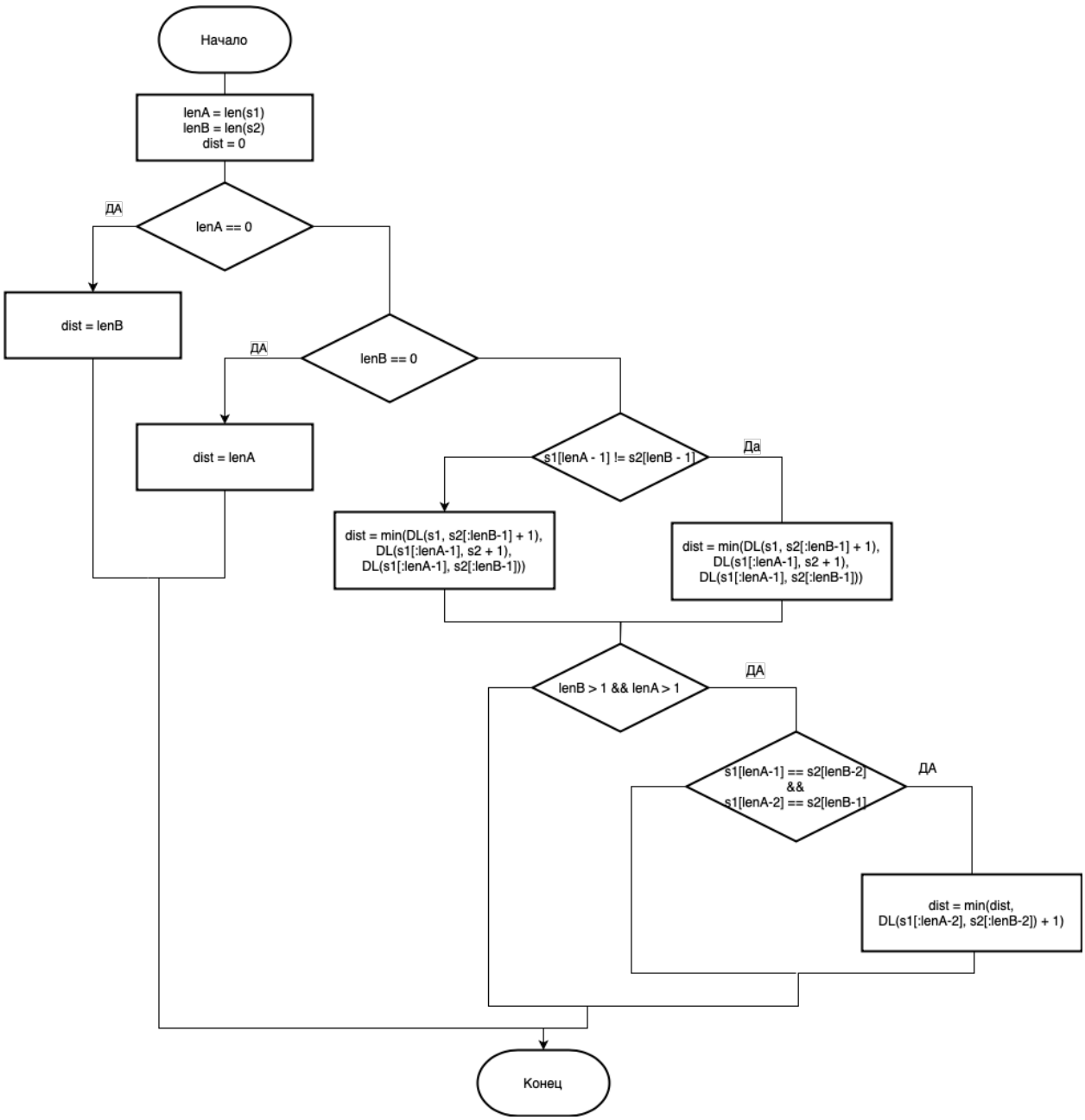

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

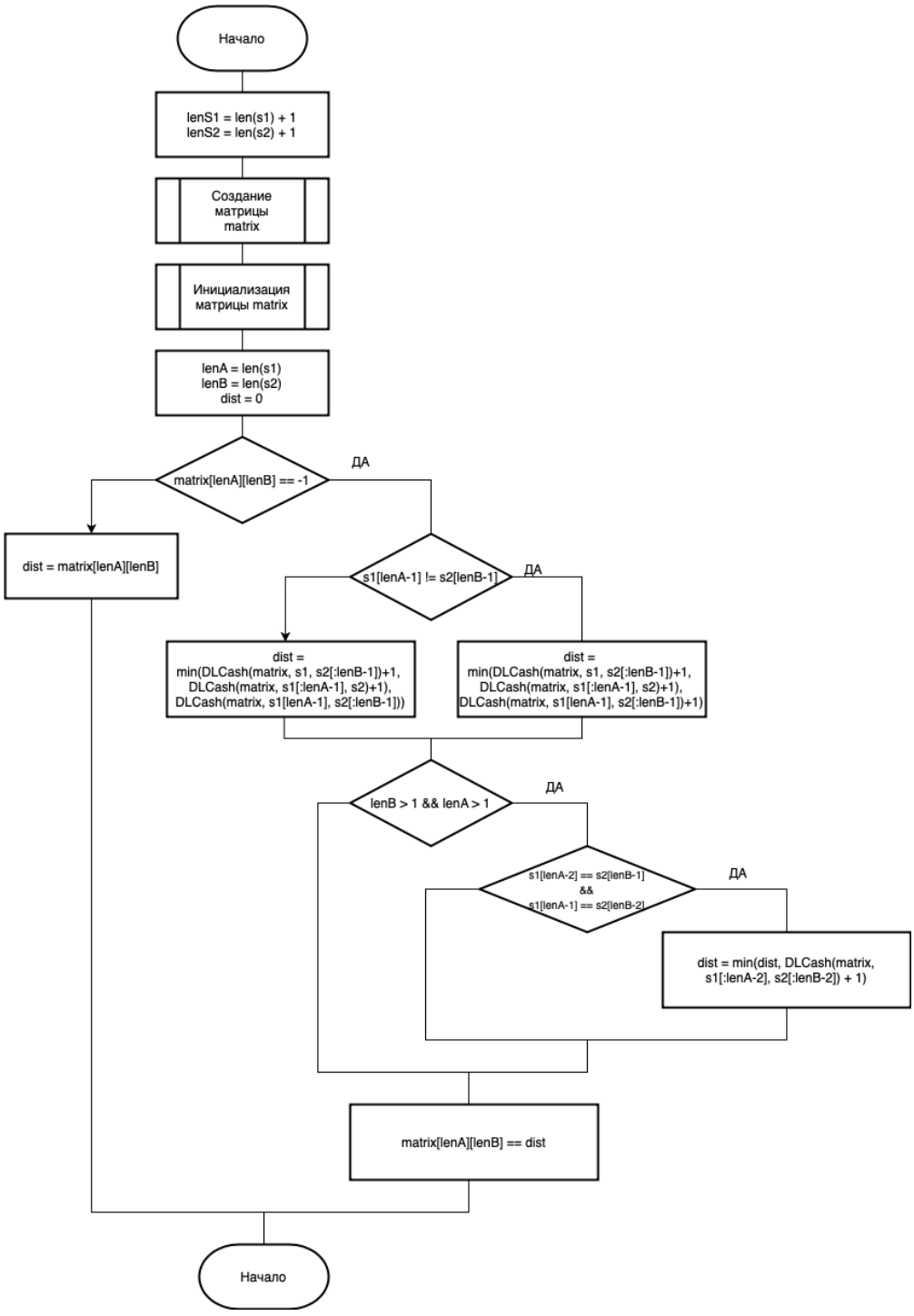

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кэшированием

2.2 Требования к ПО

Требования к программному обеспечению представлены далее:

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

2) ПО должно выводить полученное расстояние.

2.3 Вывод

На основе теоретических даннных, полученных в аналитическом разделе были построены схемы исследуемых алгоритмов.

3 Технологический раздел

3.1 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык GO[2]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язка.

Замеры времени проводились при помощи getThreadCpuTimeNs функции, написанной на C, подключенной с помощью CGO.[3]

3.2 Реализация алгоритмов

В следующих листингах приведена реализация алгоритмов нахождения расстония Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна нерекурсивным способом

```
1  func LM(matrix [][]int, a, b int, s1, s2 []rune) int {
2      for i := 1; i < a; i++ {
3          r2 := s2[i - 1]
4          for j := 1; j < b; j++ {
5              r1 := s1[j - 1]
6              if r1 != r2 {
7                  matrix[i][j] = min(matrix[i][j - 1] + 1,
                       matrix[i - 1][j] + 1, matrix[i - 1][j - 1] +
                       1)
8              } else {
9                  matrix[i][j] = min(matrix[i][j - 1] + 1,
                       matrix[i - 1][j] + 1, matrix[i - 1][j - 1])
10             }
11         }
12     }
13
14     return matrix[a - 1][b - 1]
15 }
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна-Дамерау с помощью рекурсии

```
func DL(s1, s2 []rune) int {
    lenB := len(s1)
    lenA := len(s2)

    dist := 0

    if lenA == 0 {
        dist = lenB
    } else if lenB == 0 {
        dist = lenA
    } else {
        k := 0
        if s1[lenB - 1] != s2[lenA - 1] {
            k = 1
        }

        dist = min(DL(s1, s2[ : lenA - 1]) + 1, DL(s1[ : lenB -
           1], s2) + 1, DL(s1[ : lenB - 1], s2[ : lenA - 1]) +
           k)

        if lenB > 1 && lenA > 1 && s1[lenB - 1] == s2[lenA - 2]
           && s1[lenB - 2] == s2[lenA - 1] {
            dist = min(dist, DL(s1[ : lenB - 2], s2[ : lenA -
               2]) + 1)
        }
    }

    return dist
}
```

Листинг 3.3 – Функция нахождения расстояния Левенштейна-Дамерау с помощью рекурсии и кэша

```
func DLCash(matrix [][]int, s1, s2 []rune) int {
    lenB := len(s1)
    lenA := len(s2)

    dist := 0

    if matrix[lenA][lenB] == -1 {
```

```
8           k := 0
9           if s1[lenB - 1] != s2[lenA - 1] {
10              k = 1
11          }
12
13          dist = min(DLCash(matrix, s1, s2[ : lenA - 1]) + 1,
               DLCash(matrix, s1[ : lenB - 1], s2) + 1,
               DLCash(matrix, s1[ : lenB - 1], s2[ : lenA - 1]) + k)
14
15          if lenB > 1 && lenA > 1 && s1[lenB - 1] == s2[lenA - 2]
               && s1[lenB - 2] == s2[lenA - 1] {
16              dist = min(dist, DLCash(matrix, s1[ : lenB - 2],
                   s2[ : lenA - 2]) + 1)
17          }
18
19          matrix[lenA][lenB] = dist
20      } else {
21          dist = matrix[lenA][lenB]
22      }
23
24      return dist
25  }
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна нерекурсивным способом

```
1  func DLM(matrix [][]int, a, b int, s1, s2 []rune) int {
2      for i := 1; i < a; i++ {
3          for j := 1; j < b; j++ {
4              if s1[j - 1] != s2[i - 1] {
5                  matrix[i][j] = min(matrix[i][j - 1] + 1,
                       matrix[i - 1][j] + 1, matrix[i - 1][j - 1] +
                       1)
6              } else {
7                  matrix[i][j] = min(matrix[i][j - 1] + 1,
                       matrix[i - 1][j] + 1, matrix[i - 1][j - 1])
8              }
9
10             if i > 2 && j > 2 && s1[j - 1] == s2[i - 2] && s1[j
                   - 2] == s2[i - 1] {
11                 matrix[i][j] = min(matrix[i][j], matrix[i -
                       2][j - 2] + 1)
```

```
12                 }
13             }
14         }
15
16         return matrix[a - 1][b - 1]
17 }
```

3.3 Тестирование

В таблице 3.1 представлены функциональные тесты.

Таблица 3.1 – Тестовые данные

Входные данные		**Расстояние и алгоритм**			
Строка 1	**Строка 2**	**Левенштейна**	**Дамерау-Левенштейна**		
		Итеративный	**Итеративный**	**Рекурсивный**	
				Без кэша	**С кэшом**
a	d	1	1	1	1
a	a	0	0	0	0
переговоры	перегрел	5	5	5	5
cat	cats	1	1	1	1
кот	кто	2	1	1	1
12345	54321	4	4	4	4

Вывод

Были реализованы алгоритм итеративного поиска расстояния Левенштейна, а также алгоритмы итеративного, рекурсивного и рекурсивного с кэшированием поиска редакционного расстояния Дамерау–Левенштейна. Проведено тестирование реализаций алгоритмов.

4 Исследовательская часть

4.1 Пример работы

В листинге 4.1 приведен пример работы программы.

Листинг 4.1 – Функция нахождения расстояния Дамерау-Левенштейна нерекурсивным способом

```
ivanmamvriyskiy@MacBook-Pro-Ivan-2 main % go run main.go
Меню:
1)Поиск дистанции;
2)Проверка времени;
3)Выход.
Ввод:
1

кабан
бааан
Матрица Левенштейна: 2
Матрица Дамерау-Левенштейна: 2
Рекурсивный алгоритм Дамерау-Левенштейна: 2
Рекурсивный алгоритм Дамерау-Левенштейна с кэшем: 2
```

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее:

— Процессор — 2 Гц 4-ядерный процессор Intel Core i5;

— Оперативная память — 16 ГБайт;

— Операционная система — macOS Venura 13.5.2.

4.3 Время выполнения алгоритмов

Результаты эксперимента замеров по времени приведены в таблице 4.1. В данной таблице присутствуют поля с «-». Это обусловлено тем, что для рекурсивной реализации алгоритма достаточно приведенных замеров для построения графика. При длине строки большей 10 замер времени для рекурсивного алгоритма будет долгим. Замеры проводились на одинаковых длин строк от 1 до 10 вместе с рекурсивным, от 1 до 1000 без рекурсивного.

Таблица 4.1 – Замер времени работы алгоритмов в нс

Длина строк	Нерекурсив	Нерекурсив Д. Л.	Рекурсив с кэш.	Рекурсив
1	600	500	500	0
2	700	500	600	1 000
3	700	700	2 600	2 000
4	1 000	900	1 500	6 000
5	1 100	600	1 100	20 000
6	1 200	1 400	3 200	124 000
7	1 600	1 900	3 500	568 000
8	1 600	3 300	3 700	2 881 000
9	1 700	4 400	2 900	13 157 000
10	2 300	4 100	3 100	44 023 000
100	61 400	101 100	193 400	-
200	251 900	286 700	894 300	-
300	497 800	572 900	223 3400	-
400	904 100	1 165 600	44 19 400	-
500	1 482 400	1 206 200	5 922 400	-
600	1 669 200	1 896 700	9 061 400	-
700	2 655 800	2 774 200	12 027 300	-
800	3 376 800	3 298 500	17 409 200	-
900	4 764 800	4 630 900	21 684 300	-

На основе табличных данных построены графики 4.1 и 4.2 зависимости времени выполнения алгоритма каждой из реализаций от длины строк.

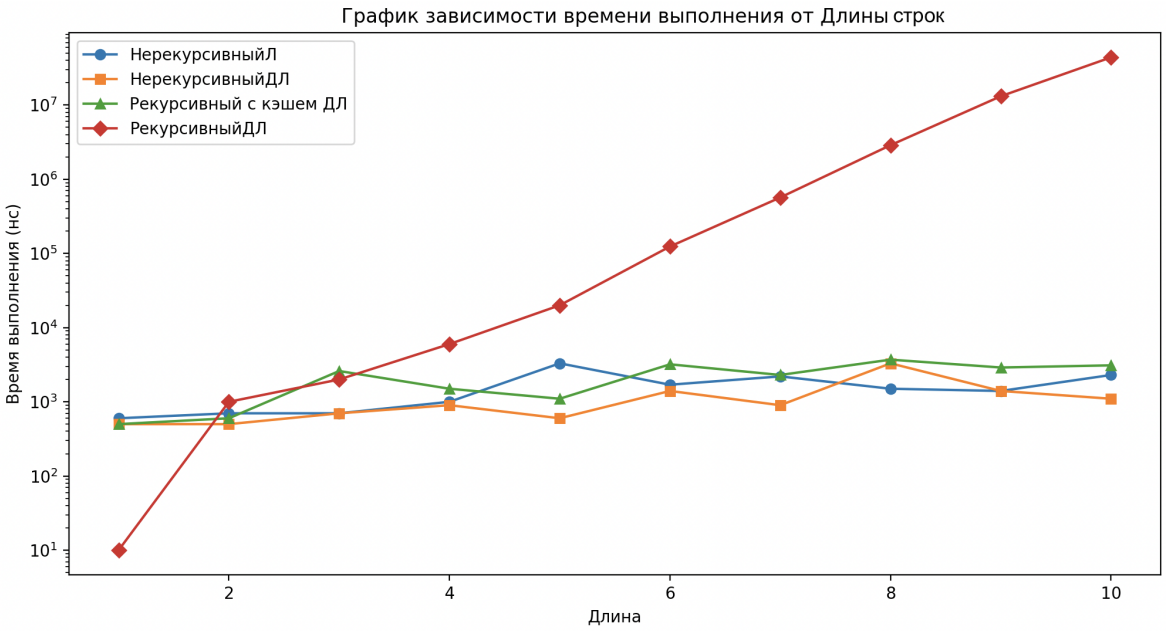

Рисунок 4.1 – График работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна

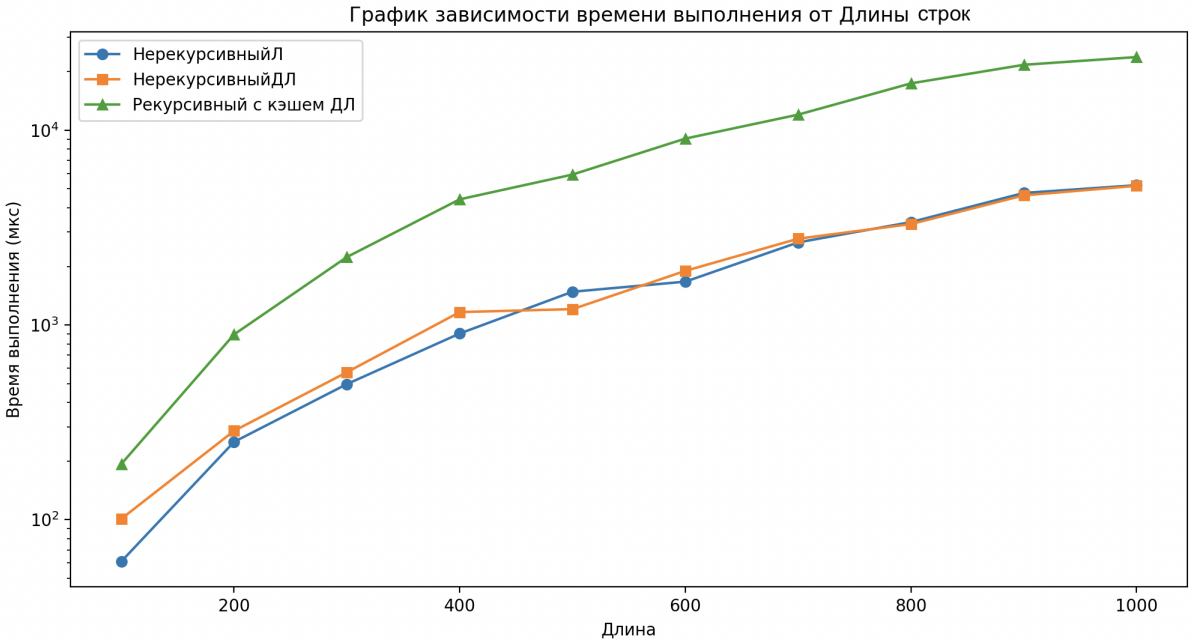

Рисунок 4.2 – График работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна

Рекурсивный алгоритм поиска расстояния Левенштейна работает в 14 — 40 тысяч раз дольше алгоритмов, использующих матрицу, уже при значении длин строк, равных 10, что свидельствует о неэффективности этого алгоритма по времени. При длинах строк менее 200 символов разница по времени между итеративными реализациями и рекурсивной с кэшем незначительна, однако при увеличении длины строки алгоритм рекурсивного поиска расстояния Дамерау-Левенштейна с кэшем оказывается медленнее вплоть до 4 раз(при длинах строк от 700).

Если рассматривать итерационные алгоритмы, то алгоритм Левенштейна работает немного быстрее, чем алгоритм Дамерау-Левенштейна, так как во втором есть дополнительная проверка.

4.4 Использование памяти

Введем следующие обозначения:

— n — длина строки S_1;

— m — длина строки S_2;

— $size()$ — функция вычисляющая размер в байтах;

— $string$ — строковый тип;

— int — целочисленный тип;

— $size\_t$ — беззнаковый целочисленный тип.

Максимальная глубина стека вызовов при рекурсивной реализации нахождения расстояния Дамерау-Левенштейна равна сумме входящих строк, а на каждый вызов требуется 2 дополнительные переменные, соответственно, максимальный расход памяти равен:

$$(n+m) \cdot (2 \cdot size(string) + 3 \cdot size(int) + 2 \cdot sizeof(size\_t)), \qquad (4.1)$$

где:

— $2 \cdot size(string)$ — хранение двух строк;

— $2 \cdot size(size\_t)$ — хранение размеров строк;

— $2 \cdot size(int)$ — дополнительные переменные;

— $size(int)$ — адрес возврата.

Для рекурсивного алгоритма с кэшированием поиска расстояния Дамерау-Левенштейна будет теоретически схож с расчетом в формуле (4.1), но также учитывается матрица, соответственно, максимальный расход памяти равен:

$$\begin{aligned}(n+m) \cdot (2 \cdot size(string) + 3 \cdot size(int) + 2 \cdot size(size\_t)) + \\ + (n+1) \cdot (m+1) \cdot size(int)\end{aligned} \qquad (4.2)$$

Использование памяти при итеративной реализации алгоритма поиска расстояния Левенштейна теоретически равно:

$$\begin{aligned}(n+1)\cdot(m+1)\cdot size(int) + 2\cdot size(string) + 2\cdot size(size\_t) + \\ + size(int**) + (n+1)\cdot size(int*) + 2\cdot size(int),\end{aligned} \tag{4.3}$$

где

— $2 \cdot size(string)$ — хранение двух строк;

— $2 \cdot size(size\_t)$ — хранение размеров матрицы;

— $(n+1)\cdot(m+1)\cdot size(int)$ — хранение матрицы;

— $size(int**) + (n+1)\cdot size(int*)$ — указатель на матрицу;

— $size(int)$ — дополнительная переменная для хранения результата;

— $size(int)$ — адрес возврата.

Использование памяти при итеративной реализации алгоритма поиска расстояния Дамерау-Левенштейна теоретически равно:

$$\begin{aligned}(n+1)\cdot(m+1)\cdot size(int) + 2\cdot size(string) + 2\cdot size(size\_t) + \\ + size(int**) + (n+1)\cdot size(int*) + 3\cdot size(int),\end{aligned} \tag{4.4}$$

где

— $2 * size(string)$ — хранение двух строк;

— $2 \cdot size(size\_t)$ — хранение размеров матрицы;

— $(n+1)\cdot(m+1)\cdot size(int)$ — хранение матрицы;

— $size(int**) + (n+1)\cdot size(int*)$ — указатель на матрицу;

— $2 \cdot size(int)$ — дополнительные переменные;

— $size(int)$ — адрес возврата.

В таблице 4.2 представлены замеры памяти при работе данных алгоритмов в зависимости от длины строк.

Таблица 4.2 – Количество используемой памяти при работе данных алгоритмов в зависимости от длины строк.

Длина строк	Нерекурсив Л	Нерекурсив ДЛ	Рекурсив кэш	Рекурсив
1	120	128	200	144
2	168	176	392	288
3	232	240	600	432
4	312	320	824	576
5	408	416	1 064	720
6	520	528	1 320	864
7	648	656	1 592	1 008
8	792	800	1 880	1 152
9	952	960	2 184	1 296
10	1 128	1 136	2 504	1 440
100	82 488	82 496	96 824	14 400
200	324 888	324 896	353 624	28 800
300	727 288	727 296	770 424	43 200
400	1 289 688	1 289 696	1 347 224	57 600
500	2 012 088	2 012 096	2 084 024	72 000
600	2 894 488	2 894 496	2 980 824	86 400
700	3 936 888	3 936 896	4 037 624	100 800
800	5 139 288	5 139 296	5 254 424	115 200
900	6 501 688	6 501 696	6 631 224	129 600
1000	8 024 088	8 024 096	8 168 024	144 000

4.5 Вывод

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

Приведенные характеристики показывают, что рекурсивная реализация алгоритма во много раз проигрывает по времени. В связи с этим, рекурсивные алгоритмы следует использовать лишь для малых размерностей строк ($<=$ 10 символов).

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

Самым затрачиваемым по памяти алгоритмом получился рекурсив-

ный алгоритм Дамерау-Левенштейна с кэшем. Итеративные алгоритмы проигрывают рекурсивному алгоритму по расходу памяти.

Заключение

В результате исследования было определено, что лучшие показатели по времени дает матричная реализация алгоритма нахождения расстояния Левенштейна, которая немного выигрывает у алгоритма нахождения расстояния Дамерау-Левенштейна за счет отсутствия дополнительной проверки. Худшие показатели у рекурсивного алгоритма нахождения расстояния Левенштейна, который проигрывает другим алгоритмам примерно в 14 – 40 тыс. раз. При этом итеративные реализации с использованием матрицы и рекурсивная с использованием кэша занимают примерно в 10 – 56 раз больше памяти, чем рекурсивная.

Цель, которая заключается в описании и исследовании алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна, выполнена, также в ходе выполнения лабораторной работы были решены следующие задачи:

1) были описаны алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна;

2) создано программное обеспечение, реализующее следующие алгоритмы:

   — нерекурсивный алгоритм поиска расстояния Левенштейна;

   — нерекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;

   — рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна;

   — рекурсивный с кешированием алгоритм поиска расстояния Дамерау-Левенштейна.

3) выбраны инструменты для замера процессорного времени выполнения реализаций алгоритмов;

4) проведен анализ затрат реализаций алгоритмов по времени и по памяти, определены влияющие на них характеристики.

Список используемых источников

1 Черненький В. М., Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. Инженерный журнал: наука и инновации, 2012. С. 30–39.

2 Документация Go [Электронный ресурс]. Режим доступа: `https://go.dev` (дата обращения: 20.09.2023).

3 Документация CGo [Электронный ресурс]. Режим доступа: `https://pkg.go.dev/cmd/cgo` (дата обращения: 20.09.2023).