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 "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Никуленко И.В.

Группа ИУ7-52Б

Оценка (баллы) _____

Преподаватель Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Целью данной лабораторной работы является изучение и реализация алгоритмов нахождения расстояний Левенштейна и Дамерау – Левенштейна

Задачами данной лабораторной являются:

- Изучение алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.
- Применение методов динамического программирования для реализации алгоритмов.
- Получение практических навыков реализации указанных алгоритмов.
- Сравнительный анализ алгоритмов на основе экспериментальных данных.
- Описание полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

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.

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:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0,\ if\ i == 0,\ j == 0; \\ i,\ if\ i > 0,\ j == 0; \\ j,\ if\ i == 0, j > 0; \\ min \begin{cases} D(s1[1..i],\ s2[1..j-1]) + 1, \\ D(s1[1..i-1],\ s2[1..j]) + 1, \\ D(s1[1..i-1],\ s2[1..j-1]) + \begin{bmatrix} 1,\ if\ s1[i] == s2[j], \\ 0,\ else \end{bmatrix} \end{cases} \end{cases} \tag{1.1}$$

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;

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:

1) сумма цены преобразования строки a в b и цены проведения операции удаления, которая необходима для преобразования a' в a;

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

3) сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются разные символы;

4) 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 1.2. В таком случае алгоритм представляет собой построчное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i, j)$.

$$D_{i,j} = min \begin{cases} (D)\ D_{i-1,j} + 1, \\ (I)\ D_{i,j-1} + 1, \\ (R)\ D_{i-1,j-1} + \begin{cases} 1, \ if\ s1[i] == s2[j]; \\ 0, \ else \end{cases} \end{cases} \tag{1.2}$$

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшированием

Рекурсивный алгоритм заполнения можно оптимизировать по времени выполнения с использованием кэширования. Кэширование - это высокоскоростной уровень хранения, на котором требуемый набор данных временного характера. [?] Благодаря наличию кэша, можно будет подставлять в

формулу уже вычисленное ранее значение, если такое имеется. Существует множество способов кэширования, а также уже готовые решения.

1.4 Расстояния Дамерау — Левенштейна

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

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

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{if } \min(i,j) = 0 \\ \min = \begin{cases} d_{a,b}(i-1,j)+1, & \\ d_{a,b}(i,j-1)+1, & \\ d_{a,b}(i-1,j-1)+m(a,b), & \text{if } \begin{cases} i,j>1; \\ a_i = b_{j-1} \\ a_{i-1} = b_j \end{cases} \\ d_{a,b}(i-2,j-2)+1, & \end{cases} & \\ \min = \begin{cases} d_{a,b}(i-1,j)+1, & \\ d_{a,b}(i,j-1)+1, & \text{else} \\ d_{a,b}(i-1,j-1)+m(a,b), & \end{cases} & \end{cases} . \tag{1.3}$$

Формула выводится по тем же соображениям, что и формула (1.3). 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

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

e

2 Конструкторская часть

2.1 Схема алгоритма Левенштейна

На рисунке 2.1 приведена схема рекурсивного алгоритма Левенштейна.

На рисунке 2.2 приведена схема рекурсивного алгоритма Левенштейна с заполнением матрицы.

На рисунке 2.3 приведена схема матричного алгоритма Левенштейна.

2.2 Схема алгоритма Дамерау — Левенштейна

На рисунке 2.4 приведена схема матричного алгоритма Дамерау – Левенштейна.

Вывод

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

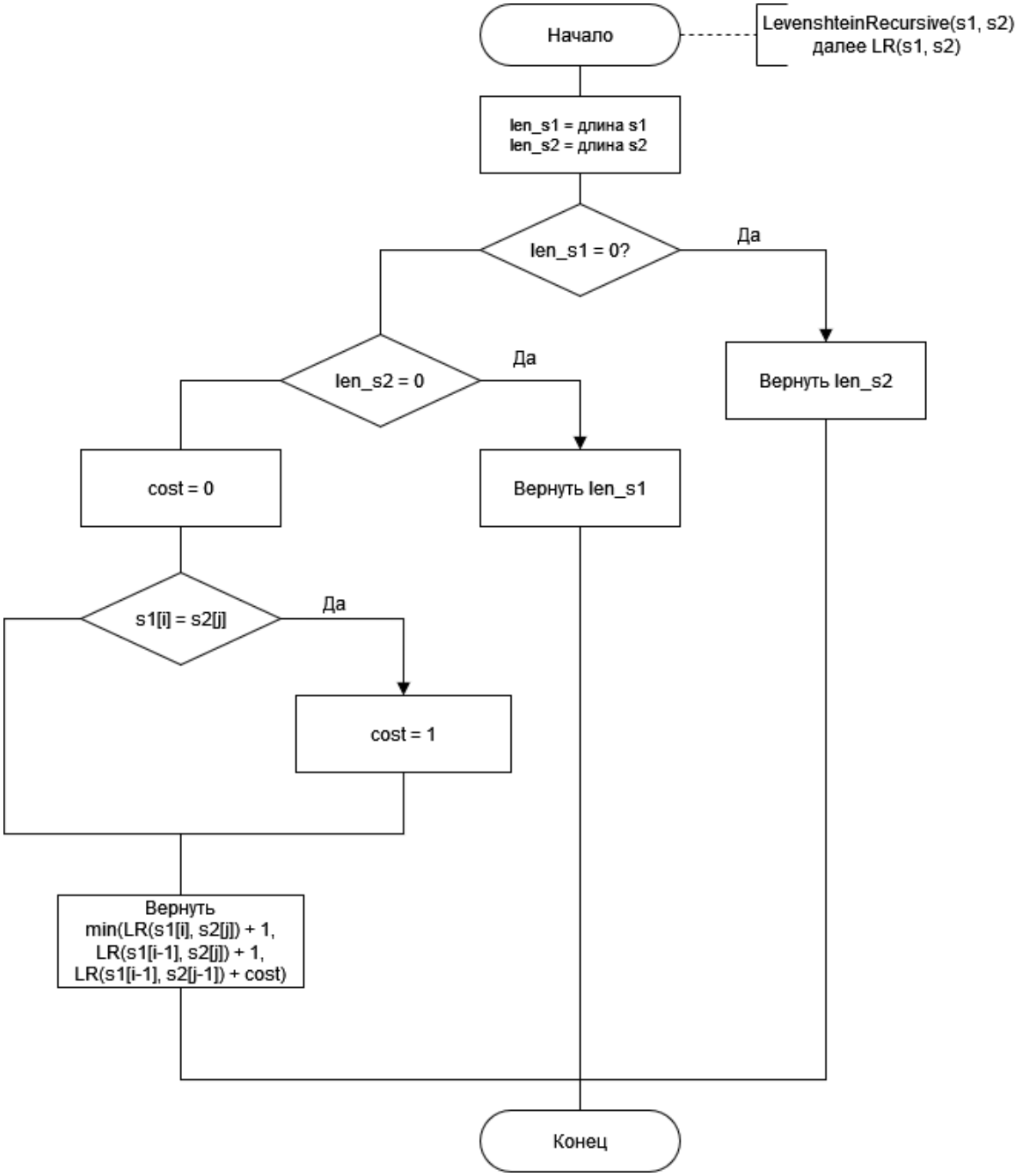

Рисунок 2.1 – Схема рекурсивного алгоритма Левенштейна

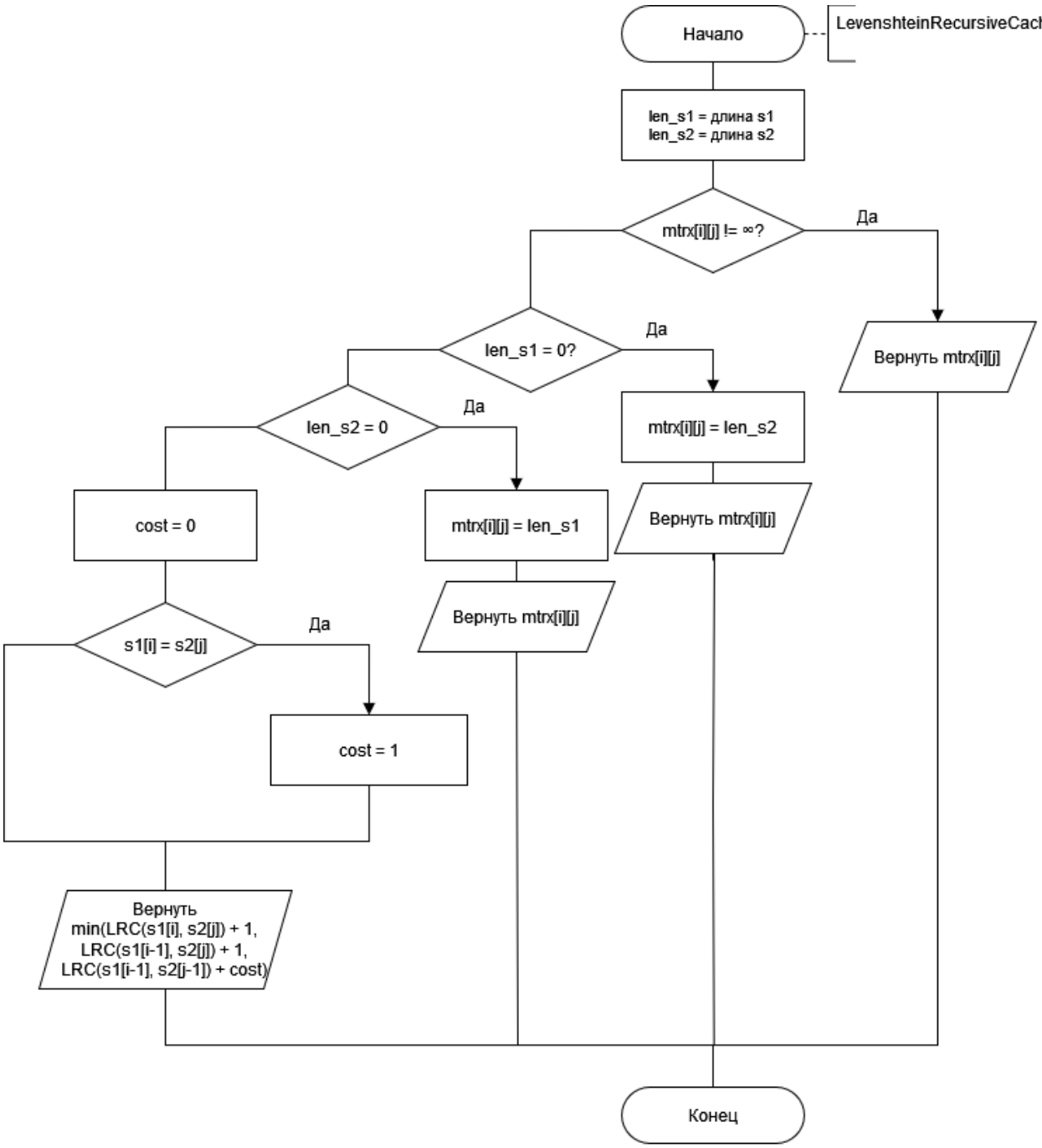

Рисунок 2.2 – Схема рекурсивного алгоритма Левенштейна с заполнением матрицы

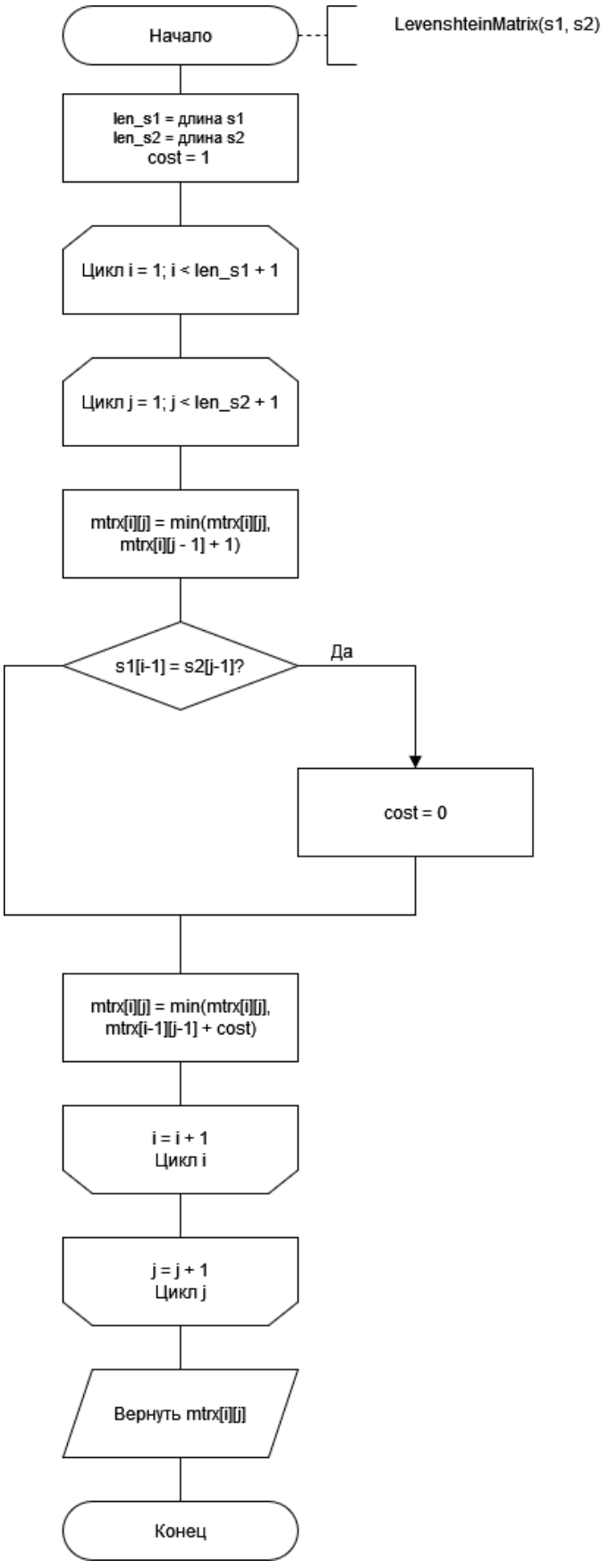

Рисунок 2.3 – Схема матричного алгоритма Левенштейна

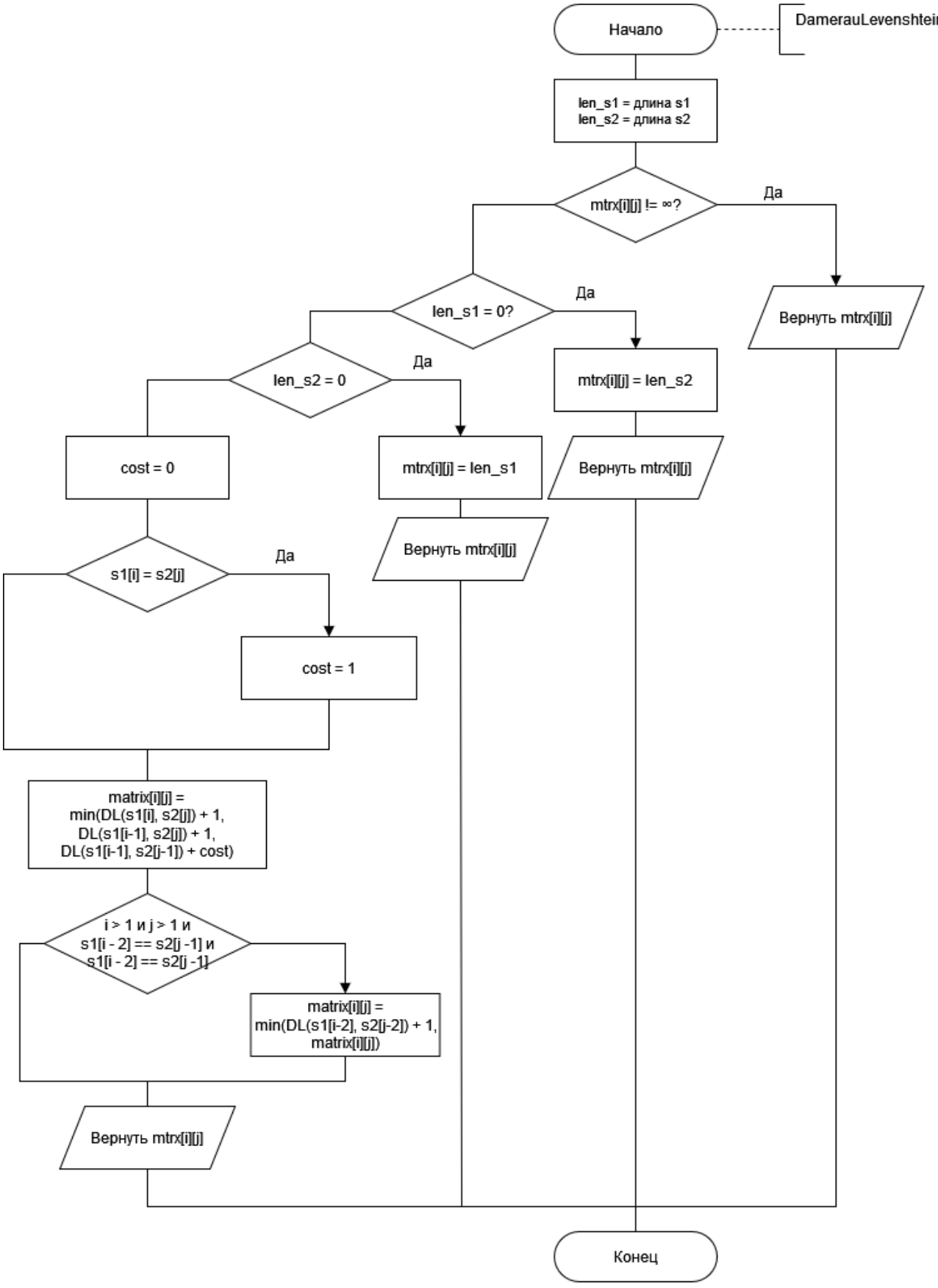

Рисунок 2.4 – Схема алгоритма Дамерау – Левенштейна

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 две строки в любой раскладке;
- на выходе — искомое расстояние для всех четырех методов.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран многопоточный язык Golang [?]. Данный выбор обусловлен моим желанием расширить свои знания в области применения данного язкыа.

3.3 Реализация алгоритма нахождения расстояния Левенштейна - рекурсивно

Листинг 3.1 – Реализация алгоритма Левенштейна рекурсивным способом

```
func LevenshteinRecursive(s1, s2 string) int {
  s1Rune, s2Rune := []rune(s1), []rune(s2)

  s1_len, s2_len := len(s1Rune), len(s2Rune)

  return distance(s1Rune, s2Rune, s1_len, s2_len)

}
```

Листинг 3.2 – Вспомогательная функция distance

```
func distance(s1, s2 []rune, i, j int) int {
  if i == 0 {
    return j
  }
  if j == 0 {
    return i
  }
  cost := 1
  if s1[i - 1] == s2[j - 1] {
    cost = 0
  }
  return findMin(distance(s1, s2, i, j - 1) + 1,
    distance(s1, s2, i - 1, j) + 1,
    distance(s1, s2, i - 1, j - 1) + cost)
}
```

3.4 Реализация алгоритма нахождения расстояния Левенштейна - матрично

Листинг 3.3 – Реализация алгоритма Левенштейна матричным способом

```
func LevenshteinMatrix(s1, s2 string) (int, intMatrix) {
  var (
    len_s1, len_s2, dist, shDist int
  )

  s1Rune, s2Rune := []rune(s1), []rune(s2)

  len_s1, len_s2 = len(s1Rune), len(s2Rune)

  distMat := make(intMatrix, len_s1+1)

  for i := range distMat {
    distMat[i] = make([]int, len_s2+1)
  }

  for i := 0; i < len_s2+1; i++ {
    distMat[0][i] = i
  }

  for i := 0; i < len_s1+1; i++ {
    distMat[i][0] = i
  }
```

```
23
24    for i := 1; i < len_s1+1; i++ {
25      for j := 1; j < len_s2+1; j++ {
26        cost := 1
27        if s1Rune[i-1] == s2Rune[j-1] {
28          cost = 0
29        }
30        dist = findMin( distMat[i][j-1] + 1,
31                distMat[i-1][j] + 1,
32                distMat[i-1][j-1] + cost)
33        distMat[i][j] = dist
34      }
35    }
36
37    shDist = distMat[len_s1][len_s2]
38
39    return shDist, distMat
40  }
```

3.5 Реализация алгоритма нахождения расстояния Левенштейна - рекурсивно с использованием кэша

Листинг 3.4 – Реализация алгоритма Левенштейна рекурсивным способом с использованием кэширования

```
1  func LevenshteinRecursiveCache(s1, s2 string) (int, intMatrix) {
2    var (
3      len_s1, len_s2, shDist int
4    )
5
6    s1Rune, s2Rune := []rune(s1), []rune(s2)
7
8    len_s1, len_s2 = len(s1Rune), len(s2Rune)
9
10   distMat := make(intMatrix, len_s1+1)
11
12   for i := range distMat {
13     distMat[i] = make([]int, len_s2+1)
14   }
15
16   for i := 0; i < len_s1+1; i++ {
```

```
    for j := 0; j < len_s2+1; j++ {
      distMat[i][j] = math.MaxInt16
    }
  }

  distanceRec(s1Rune, s2Rune, len_s1, len_s2, distMat)

  shDist = distMat[len_s1][len_s2]

  return shDist, distMat
}
```

Листинг 3.5 – Вспомогательная функция distanceRec

```
func distanceRec(s1, s2 []rune, i, j int, mat intMatrix) int {
  if mat[i][j] != math.MaxInt16 {
    return mat[i][j]
  }

  if i == 0 {
    mat[i][j] = j
    return mat[i][j]
  }
  if j == 0 {
    mat[i][j] = i
    return mat[i][j]
  }
  cost := 1
  if s1[i-1] == s2[j-1] {
    cost = 0
  }

  mat[i][j] = findMin(
    distanceRec(s1, s2, i, j-1, mat)+1,
    distanceRec(s1, s2, i-1, j, mat)+1,
    distanceRec(s1, s2, i-1, j-1, mat)+cost)
  return mat[i][j]
}
```

3.6 Реализация алгоритма нахождения расстояния Дамерау-Левенштейна - рекурсивно

Листинг 3.6 – Реализация алгоритма Дамерау-Левенштейна рекурсивным способом

```
func DamerauLevenshteinRecursive(s1, s2 string) (int, intMatrix) {
  var (
    len_s1, len_s2, shDist int
  )

  s1Rune, s2Rune := []rune(s1), []rune(s2)

  len_s1, len_s2 = len(s1Rune), len(s2Rune)

  distMat := make(intMatrix, len_s1+1)

  for i := range distMat {
    distMat[i] = make([]int, len_s2+1)
  }

  for i := 0; i < len_s1+1; i++ {
    for j := 0; j < len_s2+1; j++ {
      distMat[i][j] = math.MaxInt16
    }
  }

  DamerauLevenshtein(s1Rune, s2Rune, len_s1, len_s2, distMat)

  shDist = distMat[len_s1][len_s2]

  return shDist, distMat
}
```

Листинг 3.7 – Вспомогательная функция

```
func DamerauLevenshtein(s1, s2 []rune, i, j int, mat intMatrix) int {
  if mat[i][j] != math.MaxInt16 {
    return mat[i][j]
  }

  if i == 0 {
    mat[i][j] = j
    return mat[i][j]
  }
  if j == 0 {
    mat[i][j] = i
    return mat[i][j]
  }
  cost := 1
  if s1[i-1] == s2[j-1] {
    cost = 0
```

```
  }

  mat[i][j] = findMin(
    distanceRec(s1, s2, i, j-1, mat)+1,
    distanceRec(s1, s2, i-1, j, mat)+1,
    distanceRec(s1, s2, i-1, j-1, mat)+cost)

  if i > 1 && j > 1 && s1[i - 2] == s2[j - 1] && s1[i - 1] == s2[j - 2] {
    mat[i][j] = findMin(DamerauLevenshtein(s1, s2, i - 2, j - 2, mat) + 1, mat[i][j])
  }
  return mat[i][j]
}
```

3.7 Тестирование

На листинге 3.8 представлена реализация функции получения процессорного времени [?] в миллисекундах.

Листинг 3.8 – Измерение процессорного времени

```
/*
   #include <Windows.h>
   #include <stdio.h>

   double getCPUTime(void) {
    FILETIME createTime;
    FILETIME exitTime;
    FILETIME kernelTime;
    FILETIME userTime;
    if ( GetProcessTimes( GetCurrentProcess( ),
        &createTime, &exitTime, &kernelTime, &userTime ) != -1 )
    {
        SYSTEMTIME userSystemTime;
        if ( FileTimeToSystemTime( &userTime, &userSystemTime ) != -1 )
            return (double)userSystemTime.wHour * 3600000.0 +
                (double)userSystemTime.wMinute * 60000.0 +
                (double)userSystemTime.wSecond * 1000 +
                (double)userSystemTime.wMilliseconds;
    }
    return -1;
}
*/
import "C"

func GetCPU() float64 {
```

```
  result := (float64)(C.getCPUTime())
  return result
}
```

Листинг 3.9 – Генерация случайной строки заданного размера

```
const letterBytes = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
const (
  letterIdxBits = 6
  letterIdxMask = 1<<letterIdxBits - 1
)
func randStringBytesMask(n int) string {
  b := make([]byte, n)
  for i := 0; i < n; {
    if idx := int(rand.Int63() & letterIdxMask); idx < len(letterBytes) {
      b[i] = letterBytes[idx]
      i++
    }
  }
  return string(b)
}
```

Листинг 3.10 – Замер времени работы функций

```
func timeComparsion(iterationsNumber int) {
  var start, end float64
  tab := tabwriter.NewWriter(os.Stdout, 4, 30, 1, '␣', 0)
  tab.Write([]byte("Len\tRecursion\tMatrix\tWith_cache\tDamerau\t\n"))

  for i := 5; i < 11; i++ {
    string1 := randStringBytesMask(i)
    string2 := randStringBytesMask(i)
    fmt.Fprintf(tab, "%d\t", i)

    // Recursive
    start = GetCPU()
    for j := 0; j < iterationsNumber / 1000; j++ {
      LevenshteinRecursive(string1, string2)
    }
    end = GetCPU()
    fmt.Fprintf(tab, "%v\t", (end - start) / float64(iterationsNumber / 1000))

    // Matrix
    start = GetCPU()
    for j := 0; j < iterationsNumber; j++ {
      LevenshteinMatrix(string1, string2)
    }
    end = GetCPU()
    fmt.Fprintf(tab, "%v\t", (end - start) / float64(iterationsNumber))
```

```
26
27      // With_cache
28      start = GetCPU()
29      for j := 0; j < iterationsNumber; j++ {
30        LevenshteinRecursiveCache(string1, string2)
31      }
32      end = GetCPU()
33      fmt.Fprintf(tab, "%v\t", (end - start) / float64(iterationsNumber))
34
35      // Damerau
36      start = GetCPU()
37      for j := 0; j < iterationsNumber; j++ {
38        DamerauLevenshteinRecursive(string1, string2)
39      }
40      end = GetCPU()
41      fmt.Fprintf(tab, "%v\t\n", (end - start) / float64(iterationsNumber))
42
43    }
44    fmt.Println("Time in milliseconds")
45    tab.Flush()
46  }
```

Вывод

Были разработаны и протестированы спроектированные алгоритмы: вычисления расстояния Левенштейна рекурсивно, с заполнением матрицы и рекурсивно с заполнением матрицы, а также вычисления расстояния Дамерау — Левенштейна с заполнением матрицы.

4 Исследовательская часть

4.1 Пример работы

Демонстрация работы программы приведена на рисунке 4.1.

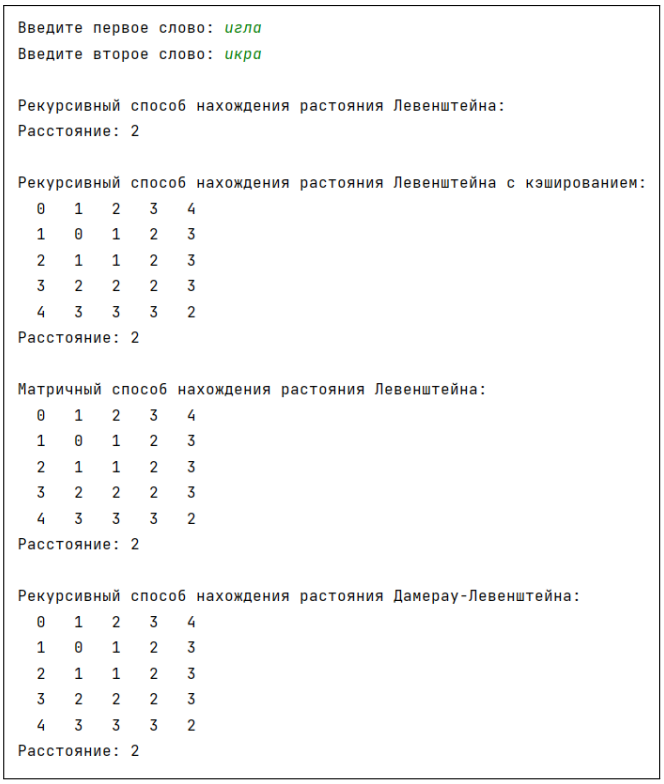

Рисунок 4.1 – Демонстрация работы алгоритмов нахождения расстояния Левенштейна и Дамерау – Левенштейна

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- Операционная система: Windows 10 Pro [?] 64-bit.
- ОЗУ: 16 GB.
- Процессор: Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz [?]

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

4.3 Время выполнения алгоритмов

Алгоритмы тестировались при помощи функции GetProcessTimes [**?**], предоставляющей возможность измерить процессорное время на программе на языке C/C++, запущенной на операционной системе Windows. Для получения более точных результатов тестирования, функции выполнялись большое количество раз. Рекурсивная реализация алгоритма вычисления расстояния Левенштейна не тестировалась при длине слов больше 10, из-за слишком большого времени работы.

Результаты замеров (в мс) приведены на рисунке 4.2. На рисунках 4.3, 4.4, **??** приведены зависимости времени работы алгоритмов от длины строк.

Time in milliseconds

Len	Recursion	Matrix	With_cache	Damerau
5	0.016	0.000906	0.001297	0.001281
6	0.078	0.001235	0.00175	0.001797
7	0.453	0.001297	0.002281	0.002219
8	2.484	0.001672	0.002969	0.004218
9	13.438	0.002109	0.003422	0.00425
10	74.641	0.002297	0.004171	0.004235
20	-	0.007281	0.016047	0.015078
30	-	0.016453	0.032453	0.032532
40	-	0.026984	0.058812	0.057032
50	-	0.0425	0.089093	0.089454
60	-	0.05964	0.129438	0.126531
70	-	0.082609	0.173016	0.173094
80	-	0.11175	0.242	0.235312
90	-	0.136953	0.293907	0.286453
100	-	0.168593	0.368329	0.359468

Рисунок 4.2 – Замер времени для строк, размером от 5 до 100

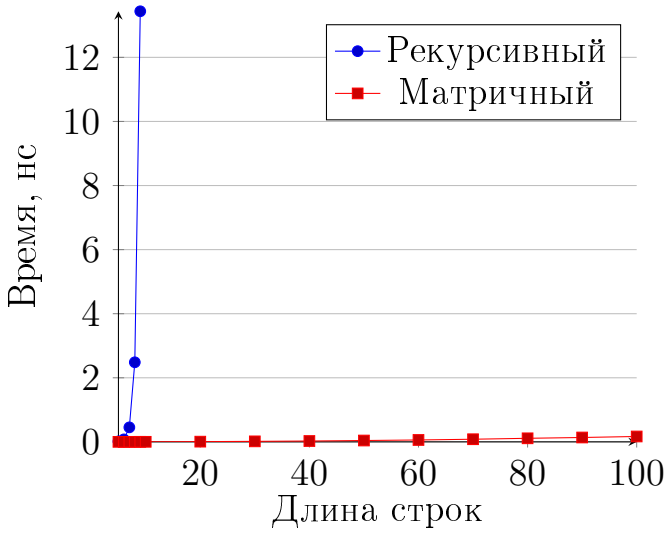

Рисунок 4.3 – Зависимость времени работы (в мс) алгоритма вычисления расстояния Левенштейна от длины строк

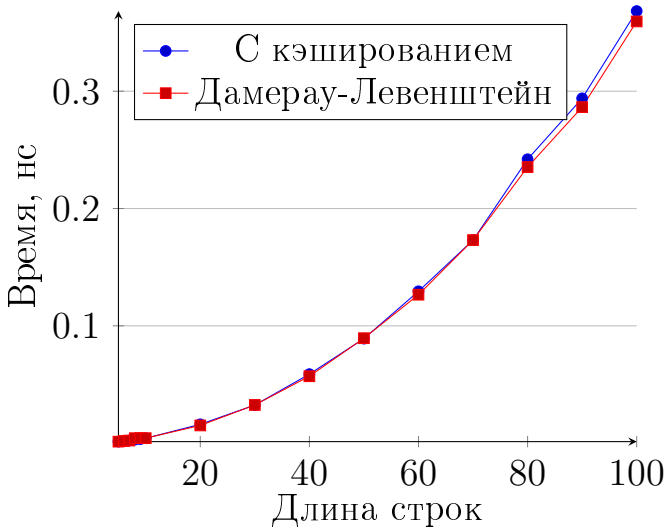

Рисунок 4.4 – Зависимость времени работы (в мс) алгоритма вычисления расстояния Левенштейна от длины строк (Матричный, С кэшированием, Дамерау-Левенштейн)

4.4 Использование памяти

Алгоритмы вычисления расстояний Левенштейна и Дамерау — Левенштейна не отличаются друг от друга с точки зрения использования памяти, следовательно, достаточно 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 вычисляется по формуле (4.1)

$$(\mathcal{C}(S_1) + \mathcal{C}(S_2)) \cdot (2 \cdot \mathcal{C}(\text{string}) + 3 \cdot \mathcal{C}(\text{size\_t})), \tag{4.1}$$

где $\mathcal{C}$ — оператор вычисления размера, S_1, S_2 — строки, int — целочисленный тип, string — строковый тип.

Использование памяти при матричной реализации теоретически вычисляется по формуле (4.2).

$$(\mathcal{C}(S_1) + 1) \cdot (\mathcal{C}(S_2) + 1) \cdot \mathcal{C}(\text{int}) + 10 \cdot \mathcal{C}(\text{int}) + 2 \cdot \mathcal{C}(\text{string})) \tag{4.2}$$

Вывод

Рекурсивный алгоритм вычисления расстояния Левенштейна работает на порядок дольше итеративных реализаций, время его работы увеличивается в геометрической прогрессии. На словах длиной 9 символов, матричная реализация алгоритма вычисления расстояния Левенштейна превосходит по времени работы рекурсивную в около 7000 раз. Рекурсивный алгоритм с заполнением матрицы превосходит простой рекурсивный на аналогичных данных в около 4500 раз. Алгоритм вычисления расстояния Дамерау — Левенштейна используется для решения других задач, поэтому говорить о его отставании от алгоритма вычисления расстояния Левенштейна, исходя из временных затрат, некорректно.

По расходу памяти алгоритмы с использованием матрицы проигрывают рекурсивному: максимальный размер используемой памяти в них растёт

как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

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. Для слов длины 9 рекурсивный алгоритм вычисления расстояния Левенштейна работает на несколько порядков медленнее (7000 раз) матричной реализации. Рекурсивный алгоритм с параллельным заполнением матрицы работает быстрее простого рекурсивного, но все еще медленнее матричного (4500 раз). Если длина сравниваемых строк превышает 9, рекурсивный алгоритм становится неприемлимым для использования по времени выполнения программы. Алгоритм вычисления расстояния Дамерау — Левенштейна используется для решения других задач, поэтому говорить о его отставании от алгоритма вычисления расстояния Левенштейна, исходя из временных затрат, некорректно.

Теоретически было рассчитано использования памяти в каждом из алгоритмов вычисления расстояния Левенштейна. Матричные алгоритмы потребляют намного больше памяти, чем рекурсивные, за счет дополнительного выделения памяти под матрицы и большего количество локальных переменных.