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"

Студент Недолужко Денис Вадимович

Группа ИУ7-53Б

Название предприятия МГТУ им. Н. Э. Баумана, каф. ИУ7

Тема Расстояние Левенштейна

Студент ____________ Недолужко Д. В.
(Подпись, дата) (И.О. Фамилия)

Руководитель курсовой работы ____________ Волкова Л. Л.
(Подпись, дата) (И.О. Фамилия)

Москва

2021 г.

Содержание

Введение

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая по модулю 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. Широко используется в теории информации и компьютерной лингвистике.

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

Расстояние Левентшейна используется:

- в компьютерной лингвистике для проведения автозамены
- в биоинформатике для сравнения генов, хромосом, белков
- в утилите diff для определения изменений в текстовых файлов

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 работы является исследование алгоритмов Левенштейна и Дамеран - Левенштейна.

Для достижения поставленной цели необходимо выполнить следующие задачи:

- изучение алгоритма Левенштейна и Демаран - Левенштейна

— реализация рекурсивных (без кеша) и итеративных (с кешем) алгоритмов Левенштейна и Дамеран - Левенштейна

— замер и сравнение процессорного времени затрачиваемых реализованными алгоритмами

— замер и сравнение используемой памяти реализованными алгоритмами

1 Аналитическая часть

Расстояние редакционное, расстояние Левеншейна - это минимальное количество операций, необходимое для превращений одной строки в другую.

Для определения стоимости преобразований вводятся следующие операции со величиной штрафа - 1:

— I (от англ. insert) - вставка

— D (от англ. delete) - удаление

— R (от англ. replace) - замена

Также вводится обозначение M (от англ. match). Штраф для совпадении двух символом считается равным 0.

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:

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

а функция 1.2 определена как:

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

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

а) для перевода из пустой строки в пустую требуется ноль операций;
б) для перевода из пустой строки в строку a требуется $|a|$ операций;
в) 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:

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

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

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

г) 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

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

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

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

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

$$D(i,j) = \begin{cases} 0, & i = 0,\ j = 0 \\ i, & i > 0,\ j = 0 \\ j, & i = 0,\ j > 0 \\ \min\{ & \\ \qquad D(i, j-1) + 1, & \\ \qquad D(i-1, j) + 1, & i > 0,\ j > 0 \\ \qquad D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ \} & \end{cases} \tag{1.4}$$

2 Конструкторская часть

2.1 Блок схемы алгоритмов

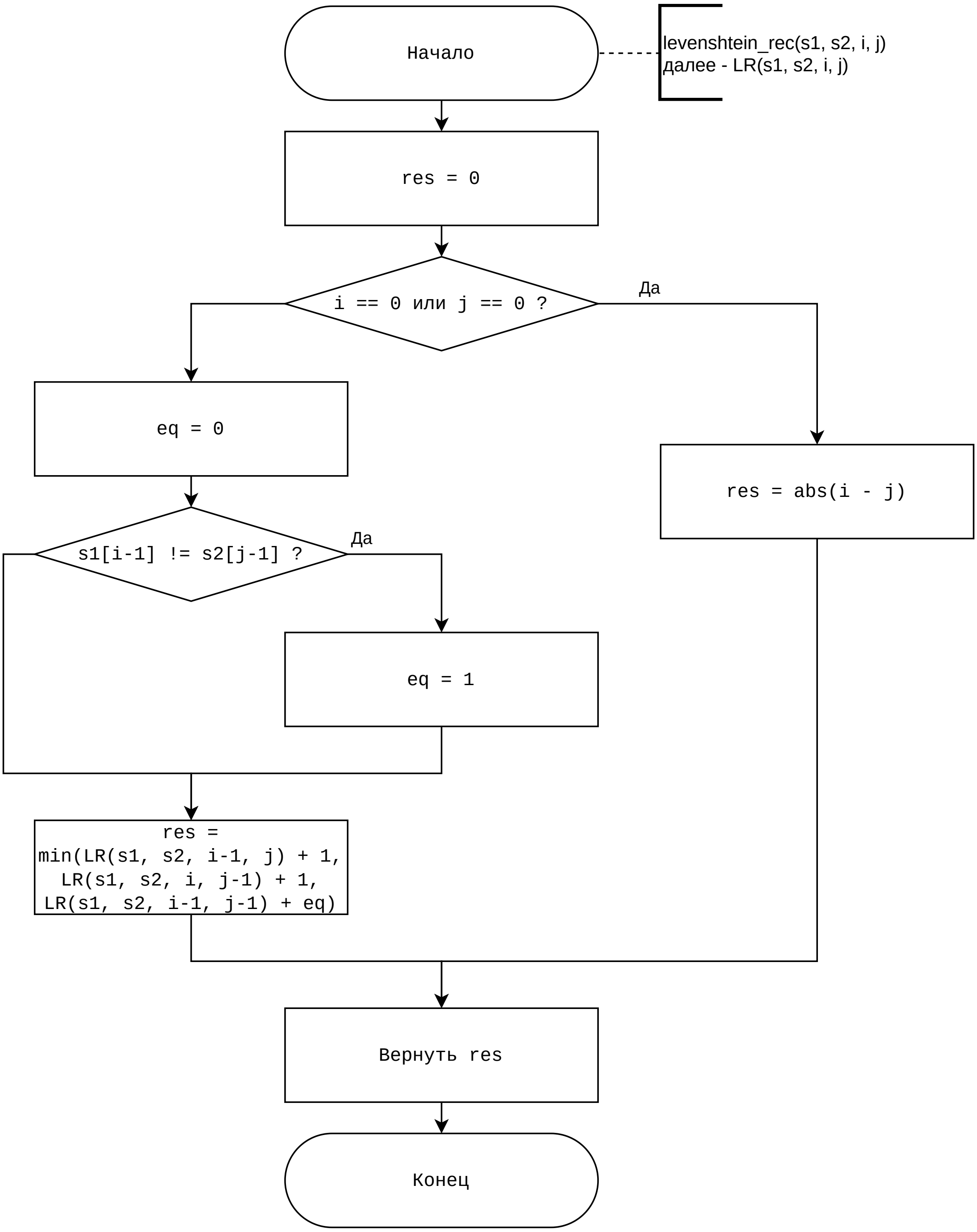

Рисунок 2.1 — Рекурсивный алгоритм Левенштейна

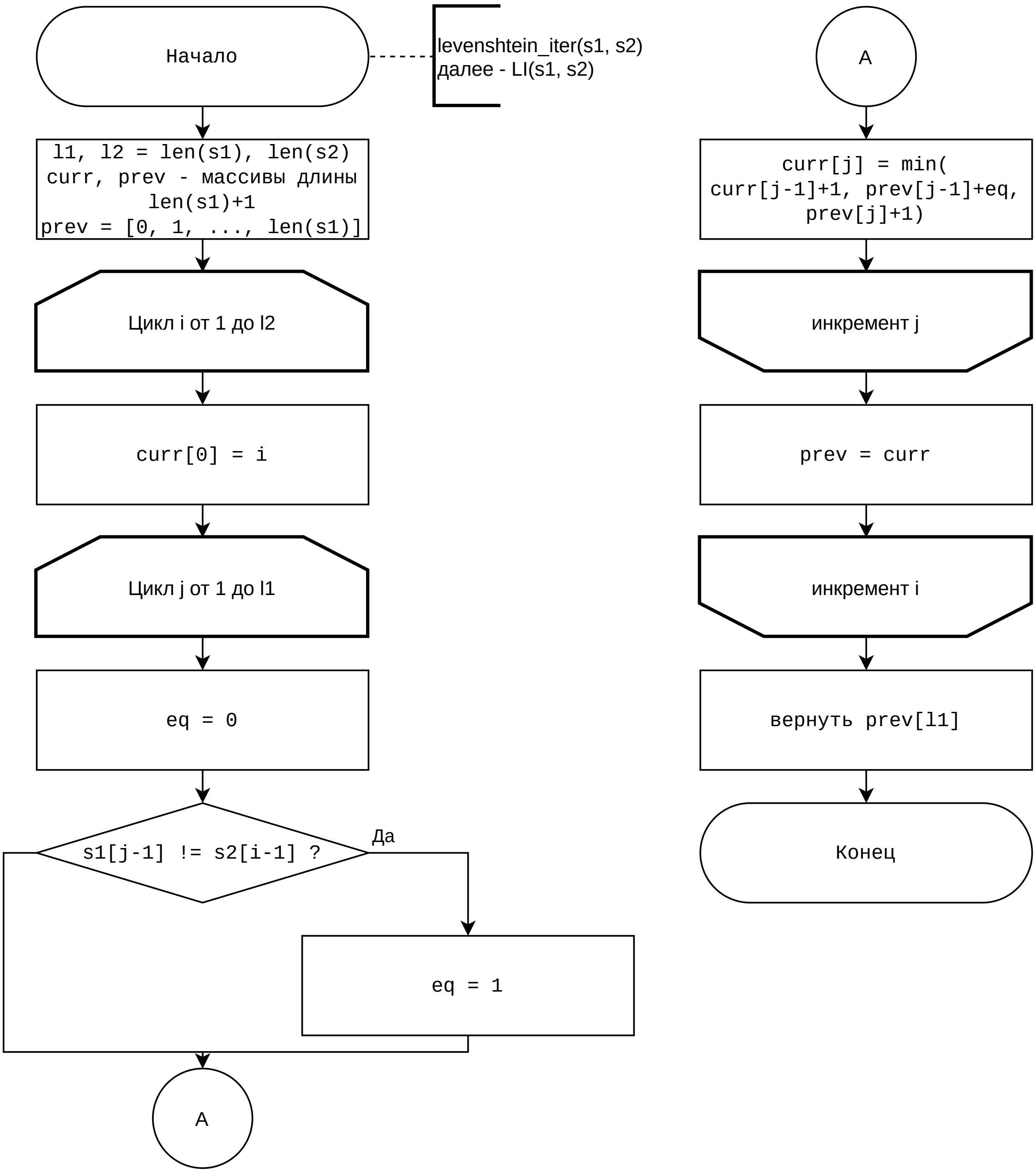

Рисунок 2.2 — Рекурсивный алгоритм Левенштейна

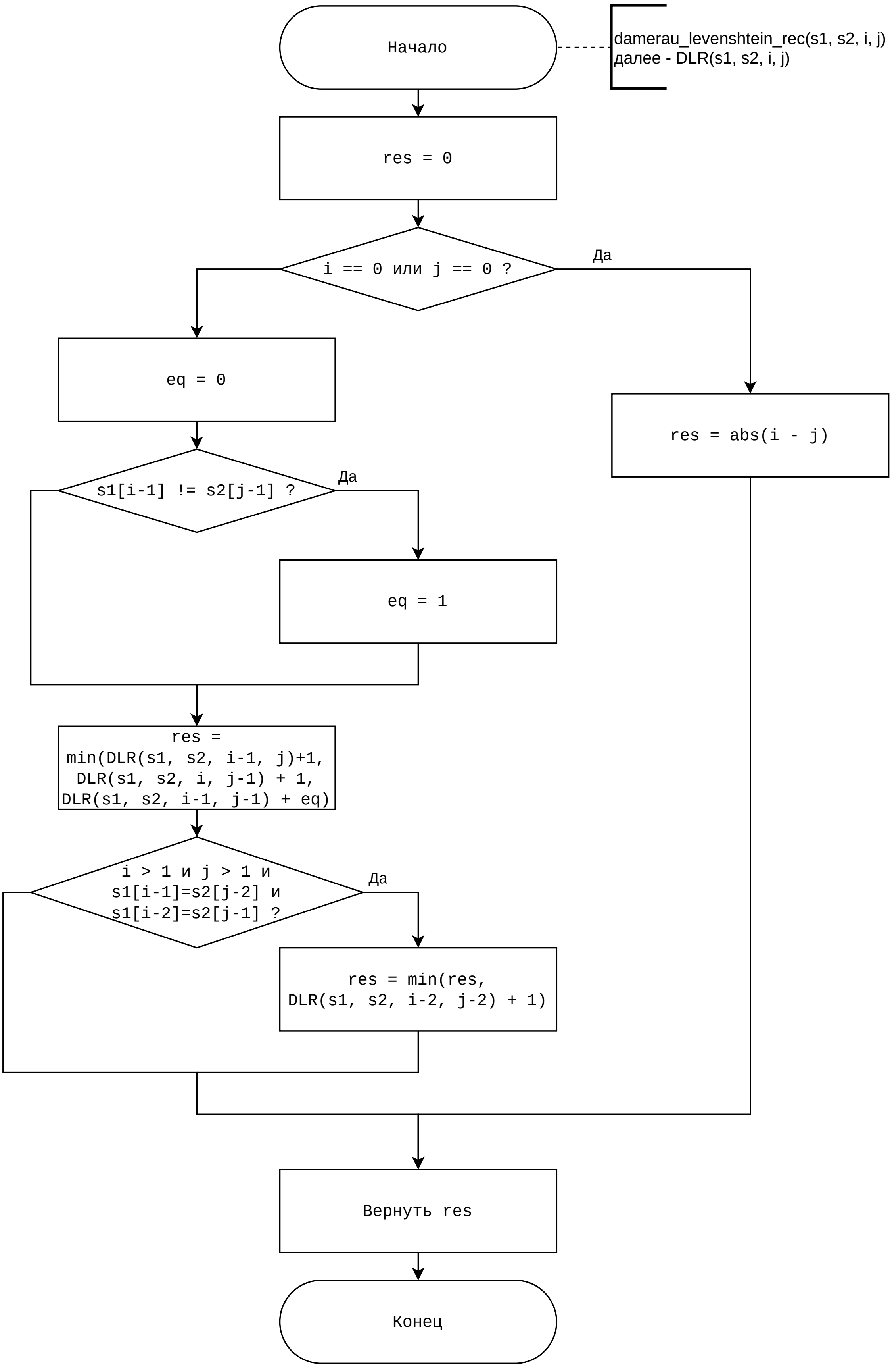

Рисунок 2.3 — Итеративный алгоритм Дамерау - Левенштейна

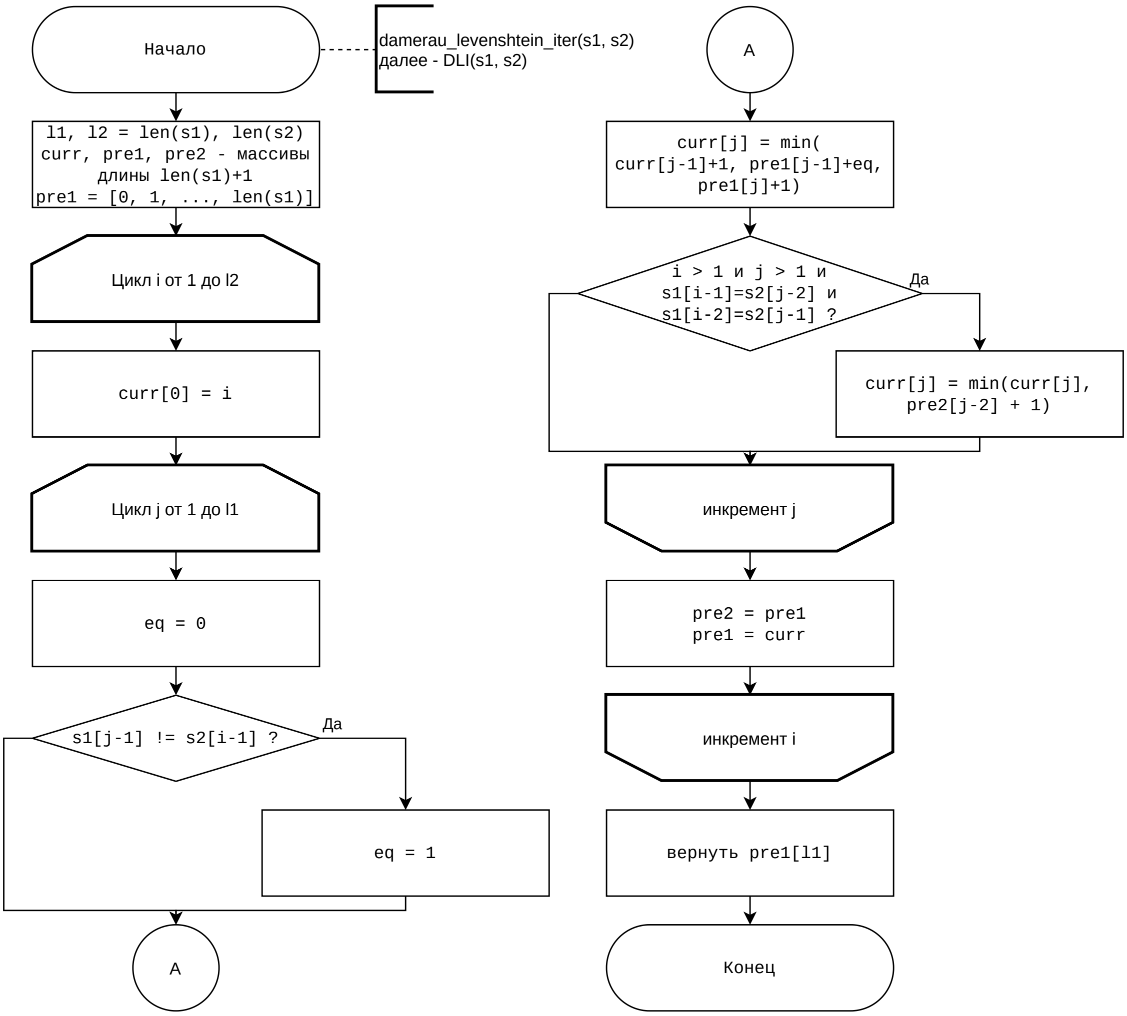

Рисунок 2.4 — Рекурсивный алгоритм Дамерау - Левенштейна

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 средств реализации

Для реализации библиотеки с реализациями алгоритмов мною был выбрал язык C стандарта 1999 года. Он был выбран в силу отсутствия необходимости в использовании высокоуровневых абстракций.

Для реализации клиента был выбран язык C расширения GNU 99 года, так как он предоставляет фукнкции чтения динамических строк.

Для юнит тестирования был выбран язык C++14, для использовая фрейворка GoogleTests[2].

3.2 Требования к программному обеспечению

К программе предъявляется ряд требований:

— На вход подаётся две регистрозависимые строки.

— На выходе — результат выполнения каждого из вышеуказанных алгоритмов.

3.3 Листинги кода

Листинг 3.1 — Основной файл программы main.c

```
#include <stdio.h>   // printf, getline
#include <stdlib.h>  // free
#include <string.h>  // strlen

#include "levenshtein/leven_dist.h"

int main(void) {
  char *s1 = NULL, *s2 = NULL;
  size_t l1 = 0, l2 = 0;

  int rc = 0;

  printf("Enter line 1: ");
  if (-1 == (rc = getline(&s1, &l1, stdin))) {
    printf("Allocation failed");
    s1 = NULL;
  } else {
    printf("Enter line 2: ");
    if (-1 == (rc = getline(&s2, &l2, stdin))) {
      printf("Allocation failed");
    } else {
      s1[strlen(s1) - 1] = '\0';
      s2[strlen(s2) - 1] = '\0';

      int res_lr = levenshtein_rec(s1, s2);
      printf("LR  ('%s', '%s') = %d\n", s1, s2, res_lr);

      int res_li = levenshtein_iter(s1, s2);
      printf("LI  ('%s', '%s') = %d\n", s1, s2, res_li);

      int res_dlr = damerau_levenshtein_rec(s1, s2);
      printf("DLR ('%s', '%s') = %d\n", s1, s2, res_dlr);

      int res_dli = damerau_levenshtein_iter(s1, s2);
      printf("DLI ('%s', '%s') = %d\n", s1, s2, res_dli);

      free(s2);
      s2 = NULL;
    }
    free(s1);
    s1 = NULL;
  }
}
```

Листинг 3.2 — Рекурсивная функция нахождения расстояния Левенштейна

```c
static int levenshtein_rec_(char const *const s1, char const *const s2,
                            int const i, int const j) {
  int res = 0;

  if (0 == i || 0 == j) {
    res = abs(i - j);
  } else {
    int eq = 0;

    if (s1[i - 1] != s2[j - 1]) eq = 1;

    res = min_int(levenshtein_rec_(s1, s2, i - 1, j) + 1,
                  min_int(levenshtein_rec_(s1, s2, i, j - 1) + 1,
                          levenshtein_rec_(s1, s2, i - 1, j - 1) + eq));
  }

  return res;
}

int levenshtein_rec(char const *const s1, char const *const s2) {
  return levenshtein_rec_(s1, s2, strlen(s1), strlen(s2));
}
```

Листинг 3.3 — Рекурсивная функция нахождения расстояния Дамерау - Левенштейна

```c
static int damerau_levenshtein_rec_(char const *const s1, char const *const s2,
                                    int const i, int const j) {
  int res = 0;

  if (0 == i || 0 == j) {
    res = abs(i - j);
  } else {
    int eq = 0;

    if (s1[i - 1] != s2[j - 1]) eq = 1;

    res = min_int(damerau_levenshtein_rec_(s1, s2, i - 1, j) + 1,
                  min_int(damerau_levenshtein_rec_(s1, s2, i, j - 1) + 1,
                          damerau_levenshtein_rec_(s1, s2, i - 1, j - 1) + eq));

    if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1]) {
      res = min_int(res, damerau_levenshtein_rec_(s1, s2, i - 2, j - 2) + 1);
    }
  }

  return res;
}

int damerau_levenshtein_rec(char const *const s1, char const *const s2) {
  return damerau_levenshtein_rec_(s1, s2, strlen(s1), strlen(s2));
}
```

Листинг 3.4 — Итеративная функция нахождения расстояния Левенштейна

```
int levenshtein_iter(char const *const s1, char const *const s2) {
  size_t l1 = strlen(s1);
  size_t l2 = strlen(s2);

  int *prev = malloc((l1 + 1) * sizeof(int));
  int *curr = malloc((l1 + 1) * sizeof(int));

  if (NULL == prev || NULL == curr) {
    if (NULL != prev) free(prev);
    if (NULL != curr) free(curr);

    return -1;
  }

  for (size_t i = 0; i <= l1; ++i) prev[i] = i;

  for (size_t i = 1; i <= l2; ++i) {
    curr[0] = i;

    for (size_t j = 1; j <= l1; ++j) {
      int eq = 0;

      if (s1[j - 1] != s2[i - 1]) eq = 1;

      curr[j] =
          min_int(curr[j - 1] + 1, min_int(prev[j - 1] + eq, prev[j] + 1));
    }

    swp_pint(&prev, &curr);
  }

  int res = prev[l1];

  free(curr);
  free(prev);

  return res;
}
```

Листинг 3.5 — Итеративная функция нахождения расстояния Дамерау - Левенштейна

```
int damerau_levenshtein_iter(char const *const s1, char const *const s2) {
  size_t l1 = strlen(s1);
  size_t l2 = strlen(s2);

  int *pre2 = malloc((l1 + 1) * sizeof(int));
  int *pre1 = malloc((l1 + 1) * sizeof(int));
  int *curr = malloc((l1 + 1) * sizeof(int));

  if (NULL == pre2 || NULL == pre1 || NULL == curr) {
    if (NULL != pre2) free(pre2);
    if (NULL != pre1) free(pre1);
    if (NULL != curr) free(curr);

    return -1;
  }

  for (size_t i = 0; i <= l1; ++i) pre1[i] = i;

  for (size_t i = 1; i <= l2; ++i) {
    curr[0] = i;

    for (size_t j = 1; j <= l1; ++j) {
      int eq = 0;

      if (s1[j - 1] != s2[i - 1]) eq = 1;

      curr[j] =
          min_int(curr[j - 1] + 1, min_int(pre1[j - 1] + eq, pre1[j] + 1));

      if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1]) {
        curr[j] = min_int(curr[j], pre2[j - 2] + 1);
      }
    }

    swp_pint(&pre2, &pre1);
    swp_pint(&pre1, &curr);
  }

  int res = pre1[l1];

  free(curr);
  free(pre1);
  free(pre2);

  return res;
}
```

Листинг 3.6 — Вспомогательные функции

```
static inline int min_int(const int l, const int r) { return l <= r ? l : r; }

static inline void swp_pint(int **p1, int **p2) {
  void *bucket = *p1;
  *p1 = *p2;
  *p2 = bucket;
}
```

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.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось исследование:

— Процессор: Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz [3]
— Оперативная память: 8 Gb
— Операционная система: elementary OS 6 Odin [4]

4.2 Тестирование

Юнит тестирование проводилось при помощи фрэймворка GoogleTests. Были выполнены следующие тесты

Таблица 4.1 — Функциональные тесты

Строка 1	**Строка 2**	**Левенштейн**	**Дамерау-Левенштейн**
CONNECT	CONEHEAD	4	
	CONEHEAD	8	8
		0	0
deniska	deniska	0	0
abc	bc	1	1
abc	ac	1	1
abc	ab	1	1
abc	xbc	1	1
abc	axc	1	1
abc	abx	1	1
CONNECT	CNNNETC		2
abcd	badc		2

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

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 разной длины

Длина строк	**Время, сек**			
	Левенштейн		**Дамерау-Левенштейн**	
	Рекурсивный	**Итеративный**	**Рекурсивный**	**Итеративный**
1	3e-08	5.3e-08	3.6e-08	6.6e-08
2	6.7e-05	3.1e-07	6.6e-05	3.8e-07
4	8.1e-05	3.2e-07	6.6e-05	3.8e-07
6	6.6e-04	4.2e-07	6.5e-05	3.8e-07
8	0.0019	5.2e-07	0.002	6.7e-07
16	-	2e-06	-	2.7e-06
32	-	8.8e-06	-	1.3e-05
64	-	3.5e-05	-	4.4e-05
128	-	0.00012	-	0.00016
256	-	0.00048	-	0.00064
512	-	0.0019	-	0.0025
1024	-	0.0075	-	0.01

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 64 символов разница по времени между итеративными реализациями незначительна, однако при увеличении длины строки алгоритм поиска расстояния Левенштейна оказывается быстрее вплоть до полутора раз. Это обосновывается тем, что у алгоритма поиска расстояния Дамерау-Левенштейна задействуется дополнительная операция, которая замедляет алгоритм.

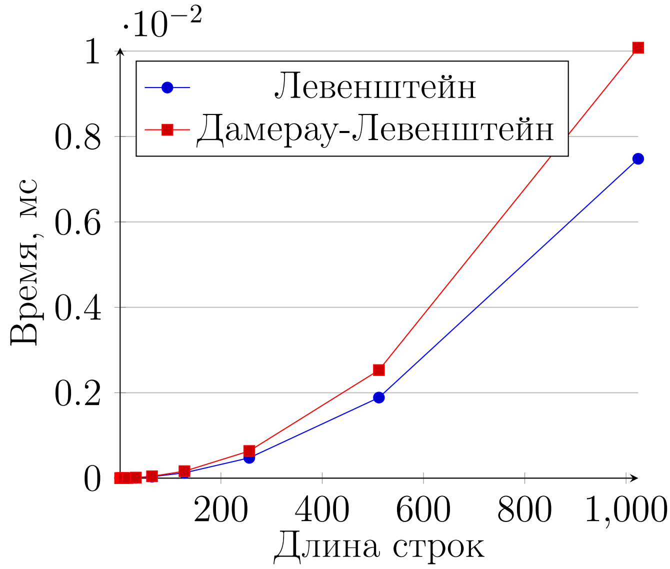

Рисунок 4.1 — Сравнение времени работы итеративных алгоритмов Левенштейна и Дамерау-Левенштейна

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

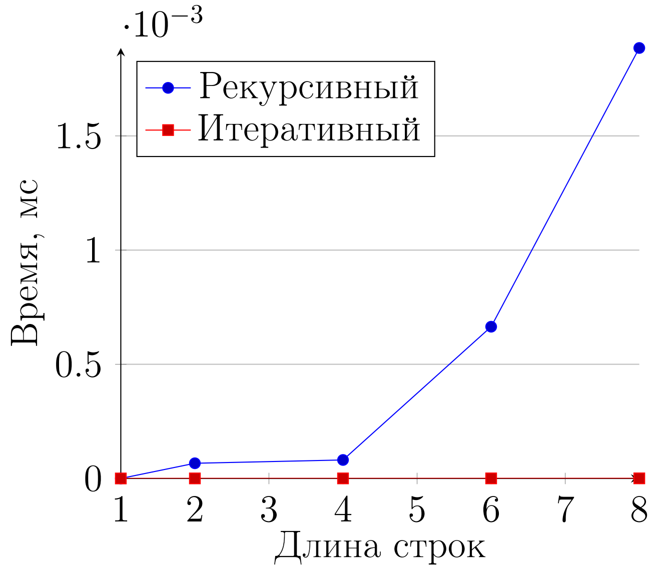

Рисунок 4.2 — Сравнение времени работы итеративного и рекурсивного алгоритмов Леверштейна

4.4 Характеристики по памяти

Алгоритмы 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 (4.1)

$$(\mathcal{C}(S_1) + \mathcal{C}(S_2)) \cdot (2 \cdot \mathcal{C}(\mathrm{string}) + 3 \cdot \mathcal{C}(\mathrm{int})), \tag{4.1}$$

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

Использование памяти при итеративной реализации теоретически равно

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

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.

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 1.1 и Дамерау – Левенштейна 1.3, построены схемы (2.2, 2.4), соответствующие данным алгоритмам, также разобраны рекурсивные алгоритмы (2.1, 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:

а) Рекурсивный алгоритм нахождения расстояния Левенштейна.

б) Итеративный алгоритм нахождения расстояния Левенштейна с кэшированием.

в) Рекурсивный алгоритм нахождения расстояния Дамерау – Левенштейна.

г) 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, Владимир Иосифович.* Двоичные коды с исправлением выпадений, вставок и замещений символов / Владимир Иосифович Левенштейн // Доклады Академии наук / Российская академия наук. — Vol. 163. — 1965. — Pp. 845–848.

2. Фреймворк Google tests. — Режим доступа: https://google.github.io/googletest/ (дата обращения: 11.10.2021).

3. Процессор Intel® Core™ i5-8250U. — 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: 11.10.2021).

4. Операциионная система elementary os. — Режим доступа: https://elementary.io/ (дата обращения: 11.10.2021).