Государственное образовательное учреждение высшего профессионального образования
«Московский государственный технический университет имени Н.Э. Баумана»
(МГТУ им. Н.Э. Баумана)

Лабораторная работа №1

по курсу «Анализ алгоритмов»

Расстояние Левенштейна и Дамерау-Левенштейна

Выполнил: Сорокин А.П., гр. ИУ7-52Б

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

Москва, 2019 г.

Оглавление

Введение

В современном мире почти каждый человек пользуются компьютером и Интернетом в частности. Люди пишут текст в документах, выполняют поиск в поисковых системах, ищут переводы слов и текстов в онлайн-словарях. В таких ситуациях человек часто делает орфографические ошибки или опечатки, и на их исправление он тратит своё время. Чтобы этого избежать, в подобных системах есть опции поиска ошибок и автоисправления. Для такой опции необходим поиск расстояния между строками по алгоритмам Левенштейна и Дамерау-Левенштейна. Также эта задача необходима и в программировании (например, для сравнения текстовых файлов или файлов кода в системах контроля версий) и в биоинформатике (например, для сравнения белков, генов и хромосом).

1. Аналитическая часть

1.1 Задачи

Цель лабораторной работы: исследовать расстояния Левенштейна и Дамерау-Левенштейна. Для достижения этой цели были поставлены следующие задачи:

- изучить алгоритмы вычисления расстояний между строками;
- применить метод динамического программирования для матричных реализаций алгоритмов;
- сравнить матричную и рекурсивную реализации алгоритмов;
- оценить эффективность каждой из реализаций по времени и памяти.

1.2 Описание алгоритмов

1.2.1 Расстояние Левенштейна

Расстояние Левенштейна определяет минимальное количество операций, необходимых для превращения одной строки в другую, среди которых:

- вставка (I - insert);
- удаление (D - delete);
- замена (R - replace).

У каждой операции есть так называемая "цена или "штраф"за её выполнение. Цена каждой операции равна 1, кроме случая совпадения символов (M - match); цена в этом случае равна 0, т. к. при равенстве символов не требуется никаких действий. Соответственно, задача нахождения расстояния Левенштейна заключается в нахождении такой последовательности операции, приводящик одну строку к другой, суммарная цена которых минимальна.

Таким образом, если заданы две строки S_1 и S_2 с длинами m и n соответственно над некоторым алфавитом, то расстояние Левенштейна $D(S_1, S_2)$ между данными строками можно вычислить по следующей рекуррентной формуле [3]:

$$D(S_1[1..m], S_2[1..n]) = \begin{cases} m \ if \ n = 0 \\ n \ if \ m = 0 \\ min \begin{cases} D(S_1[1..m-1], S_2[1..n] + 1) \\ D(S_1[1..m], S_2[1..n-1] + 1) \\ D(S_1[1..m-1], S_2[1..n-1] + (S_1[m] \neq S_2[n])) \end{cases} \end{cases} \tag{1.1}$$

Соотношения в рекурретной формуле отвечают за соотвествующие разрешённые операции:

1. Вставка.
2. Удаление.
3. Замена или совпадение в зависимости от результата $(S_1[m] \neq S_2[n])$.

1.2.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояние Левенштейна. К исходному набору возможных операций добавляется операция транспозиции (T - transpose), или перестановка двух соседних символов. В своих исследованиях Ф. Дамерау показал, что наиболее частой ошибкой при вводе текста является перестановка двух соседних букв слов [2]. "Цена"данной операции также равняется 1. При вычислении расстояния Левенштейна в такой ситуации потребовалось бы дважды заменить символ. Суммарная цена этих двух операций равнялась бы 2, а транспозиция добавляет в суммарную цену лишь 1. Исходя из этого, можно утверждать, что расстояние Дамерау-Левенштейна даёт лучший результат в сравнении с расстоянием Левенштейна.

При вычислении расстояния Дамерау-Левенштейна в рекурретную формулу вносится дополнительное соотношение в минимум:

$$D(S_1[1..m-2], S_2[1..n-2]) + 1 \tag{1.2}$$

Соотношение (1.2) вносится в выражение только при выполнении следующих условий:

$$\begin{cases} m > 2, n > 2 \\ S_1[m] = S_2[n-1] \\ S_1[m-1] = S_2[n] \end{cases} \tag{1.3}$$

Таким образом получаем следующую рекурретную формулу:

$$D(S_1[1..m], S_2[1..n]) = \begin{cases} m\ if\ n = 0 \\ n\ if\ m = 0 \\ min \begin{cases} D(S_1[1..m-1], S_2[1..n] + 1) \\ D(S_1[1..m], S_2[1..n-1] + 1) \\ D(S_1[1..m-1], S_2[1..n-1] + (S_1[m] \neq S_2[n])) \\ D(S_1[1..m-2], S_2[1..n-2]) + 1 \end{cases} & \text{if } (1.3) \\ min \begin{cases} D(S_1[1..m-1], S_2[1..n] + 1) \\ D(S_1[1..m], S_2[1..n-1] + 1) \\ D(S_1[1..m-1], S_2[1..n-1] + (S_1[m] \neq S_2[n])) \end{cases} & \text{otherwise} \end{cases} \tag{1.4}$$

2. Конструкторская часть

2.1 Схемы алгоритмов

На рисунках 2.1 - 2.3 представлены схемы алгоритмов трёх реализаций алгоритмов поиск расстояния между строками.

Рис. 2.1: Матричная реализация алгоритма Левенштейна

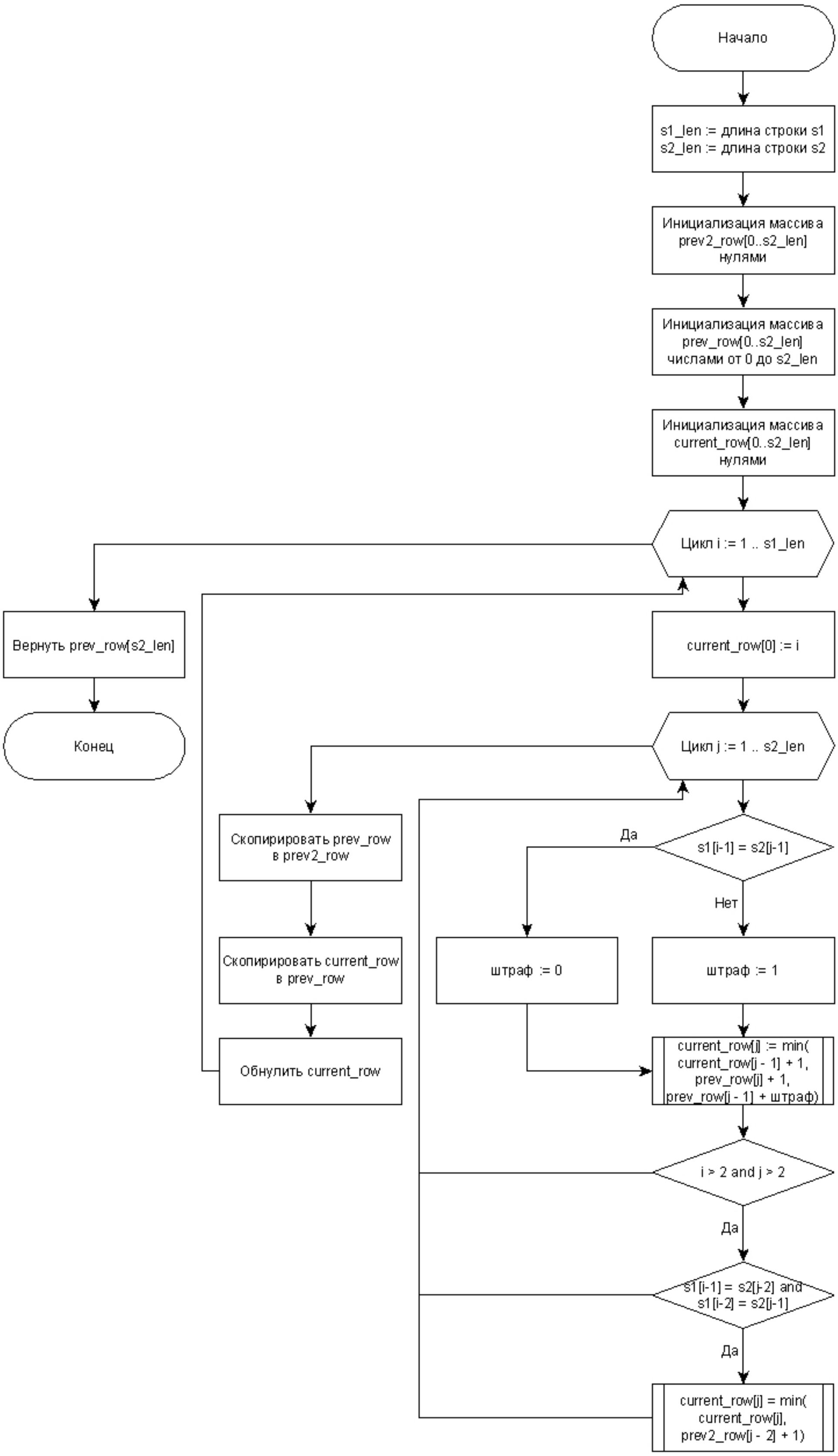

Рис. 2.2: Матричная реализация алгоритма Дамерау-Левенштейна

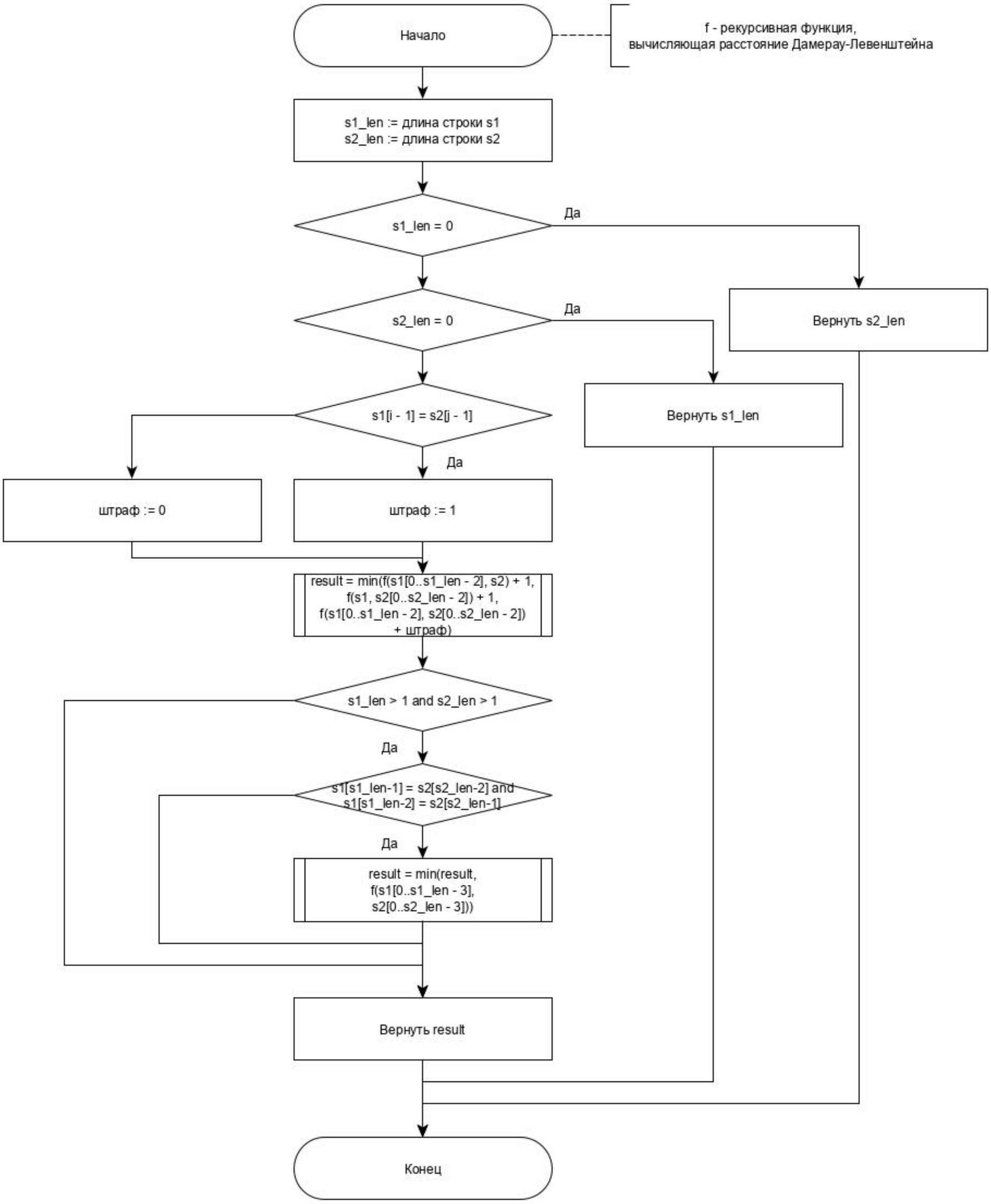

Рис. 2.3: Рекурсивная реализация алгоритма Дамерау-Левенштейна

3. Технологическая часть

3.1 Требования к программному обеспечению

На вход подаются две строки максимальной длины в 50 символов, которые входят в таблицу Юникода (UTF-8). На выход программа выдаёт три числовых значения, которые являются результатами вычисления расстояний тремя методам: матричными реализациями алгоритмов Левенштейна и Дамерау-Левенштейна и рекурсивной реализацией алгоритма Дамерау-Левенштейна. В качестве результата для матричных реализаций также выводится матрица расстояний.

3.2 Средства реализации

Для реализации программы был использован язык C++ [4]. Для замера процессорного времени была использована функция rdtsc() из библиотеки stdrin.h.

3.3 Реализации алгоритмов

На листингах 3.1 - 3.3 представлены коды реализации алгоритмов поиска расстояния.

Листинг 3.1: Матричная реализация алгоритма Левенштейна

```
unsigned levenshtein(std::string s1, std::string s2, bool to_print)
{
  size_t s1_len = s1.length(), s2_len = s2.length();
  size_t row_length = s2_len + 1;
  unsigned row_bytes = row_length * sizeof(unsigned);

  unsigned *prev_row = new unsigned[row_length];
  unsigned *current_row = new unsigned[row_length];
  for (size_t i = 0; i < row_length; i++)
  prev_row[i] = i;

  if (to_print)
  {
    for (size_t i = 0; i < row_length; i++)
      std::cout << prev_row[i] << ' ';
    std::cout << std::endl;
  }

  for (size_t i = 1; i <= s1_len; i++)
  {
    current_row[0] = i;
    for (size_t j = 0; j < row_length; j++)
```

```
    {
      unsigned match_fault = unsigned(s1[i - 1] != s2[j - 1]);
      current_row[j] = std::min({current_row[j - 1] + 1,
                                 prev_row[j] + 1,
                                 prev_row[j - 1] + match_fault});
    }
    if (to_print)
    {
      for (size_t k = 0; k < row_length; k++)
        std::cout << current_row[k] << ' ';
      std::cout << std::endl;
    }
    memcpy(prev_row, current_row, row_bytes);
  }

  unsigned result = current_row[s2_len];

  delete [] prev_row;
  delete [] current_row;

  return result;
}
```

Листинг 3.2: Матричная реализация алгоритма Дамерау-Левенштейна

```
unsigned damerau(std::string s1, std::string s2, bool to_print)
{
  size_t s1_len = s1.length(), s2_len = s2.length();
  size_t row_length = s2_len + 1;
  unsigned row_bytes = row_length * sizeof(unsigned);

  unsigned *prev2_row = new unsigned[row_length];
  unsigned *prev_row = new unsigned[row_length];
  unsigned *current_row = new unsigned[row_length];
  for (size_t i = 0; i < row_length; i++)
  {
    prev2_row[i] = 0;
    prev_row[i] = i;
  }

  if (to_print)
  {
    for (size_t i = 0; i < row_length; i++)
      std::cout << prev_row[i] << ' ';
    std::cout << std::endl;
  }

  for (size_t i = 1; i <= s1_len; i++)
  {
    current_row[0] = i;
    for (size_t j = 0; j < row_length; j++)
    {
      unsigned match_fault = unsigned(s1[i - 1] != s2[j - 1]);
      current_row[j] = std::min({current_row[j - 1] + 1,
```

```
                        prev_row[j] + 1,
                        prev_row[j - 1] + match_fault});
      if (i >= 2 && j >= 1)
        if (s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1])
          current_row[j] = std::min(current_row[j],
                                    prev2_row[j - 2] + 1);
    }

    if (to_print)
    {
      for (size_t k = 0; k < row_length; k++)
        std::cout << current_row[k] << ' ';
      std::cout << std::endl;
    }

    memcpy(prev2_row, prev_row, row_bytes);
    memcpy(prev_row, current_row, row_bytes);
  }

  unsigned result = current_row[s2_len];

  delete [] prev2_row;
  delete [] prev_row;
  delete [] current_row;

  return result;
}
```

Листинг 3.3: Рекурсивная реализация алгоритма Дамерау-Левенштейна

```
unsigned damerau_r(std::string s1, std::string s2, bool to_print)
{
  size_t s1_len = s1.length(), s2_len = s2.length();
  if (s1_len == 0)
    return s2_len;
  if (s2_len == 0)
    return s1_len;

  unsigned match_fault = unsigned(s1[s1_len - 1] != s2[s2_len - 1]);

  unsigned result = std::min({damerau_r(s1.substr(0, s1_len - 1),
                 s2.substr(0, s2_len)) + 1,
                 damerau_r(s1.substr(0, s1_len),
                 s2.substr(0, s2_len - 1)) + 1,
                 damerau_r(s1.substr(0, s1_len - 1),
                 s2.substr(0, s2_len - 1)) + match_fault});

  if (s1_len > 1 && s2_len > 1)
    if (s1[s1_len - 1] == s2[s2_len - 2] &&
      s1[s1_len - 2] == s2[s2_len - 1])
      return std::min(result, damerau_r(s1.substr(0, s1_len - 2),
                              s2.substr(0, s2_len - 2)) + 1);

  return result;
```

```
}
```

3.4 Тесты

Для проверки корректности работы были подготовлены функциональные тесты, представленные в таблице 3.1. В данной таблице λ означает пустую строку, а числа в столбцах "Ожидание"и "Результат"соответствуют результатам работы алгоритмов в следующем порядке:

1. Матричная реализация алгоритма Левенштейна.
2. Матричная реализация алгоритма Дамерау-Левенштейна.
3. Рекурсивная реализация алгоритма Дамерау-Левенштейна.

Таблица 3.1: Функциональные тесты

Строка 1	**Строка 2**	**Ожидание**	**Результат**
λ	λ	0 0 0	0 0 0
λ	a	1 1 1	1 1 1
a	λ	1 1 1	1 1 1
a	a	0 0 0	0 0 0
a	б	1 1 1	1 1 1
азы	базы	1 1 1	1 1 1
компютер	компьютер	1 1 1	1 1 1
данны	данные	1 1 1	1 1 1
email.ru	mail.ru	1 1 1	1 1 1
programmmer	programmer	1 1 1	1 1 1
mail.rus	mail.ru	1 1 1	1 1 1
ашибка	ошибка	1 1 1	1 1 1
алгоритм	алгорифм	1 1 1	1 1 1
копия	копии	1 1 1	1 1 1
укрсовой	курсовой	2 1 1	2 1 1
аглоритм	алгоритм	2 1 1	2 1 1
унивре	универ	2 1 1	2 1 1
курс	курсовой	4 4 4	4 4 4
курсовой	курс	4 4 4	4 4 4
курсовой	курсовик	2 2 2	2 2 2
код	закодировать	9 9 9	9 9 9
закодировать	код	9 9 9	9 9 9
ccoders	recoding	5 5 5	5 5 5
header	subheader	3 3 3	3 3 3
subheader	header	3 3 3	3 3 3
subheader	overheader	4 4 4	4 4 4

В результате проверки все реализации алгоритмов прошли все поставленные функциональные тесты.

4. Экспериментальная часть

4.1 Примеры работы

На рисунке 4.1 представлен пример работы программы, демонстрирующий различие в работе алгоритмов наглядно: различаются матрицы расстояний и результаты.

```
Enter first word: foledr
Enter second word: folder
Levenshtein:
0 1 2 3 4 5 6
1 0 1 2 3 4 5
2 1 0 1 2 3 4
3 2 1 0 1 2 3
4 3 2 1 1 1 2
5 4 3 2 1 2 2
6 5 4 3 2 2 2
2
Damerau-Levenshtein (matrix):
0 1 2 3 4 5 6
1 0 1 2 3 4 5
2 1 0 1 2 3 4
3 2 1 0 1 2 3
4 3 2 1 1 1 2
5 4 3 2 1 1 2
6 5 4 3 2 2 1
1
Damerau-Levenshtein (recursive):
1
```

Рис. 4.1: Пример работы программы

4.2 Сравнение работы алгоритмов Левенштейна и Дамерау-Левенштейна

Для сравнения времени работы алгоритмов Левенштейна и Дамерау-Левенштейна были использованы строки длиной от 10 до 70 с шагом 10. Эксперимент для более точного результата повторялся 100 раз. Итоговый результат рассчитывался как средний из полученных результатов. Результаты измерений показаны в таблице 4.1 и на рисунке 4.2.

Алгоритм Левенштейна выигрывает по времени в среднем не более, чем на 10%. Алгоритм Дамерау-Левенштейна выполняется дольше за счёт добавления небольшого количества операций.

Таблица 4.1: Время работы матричных реализаций алгоритмов в тактах процессора

Длина слова	Алгоритм Левенштейна	Алгоритм Дамерау-Левенштейна
10	11095	11734
20	46337	47488
30	75911	84603
40	135158	154336
50	200624	220133
60	308936	345137
70	381480	422824

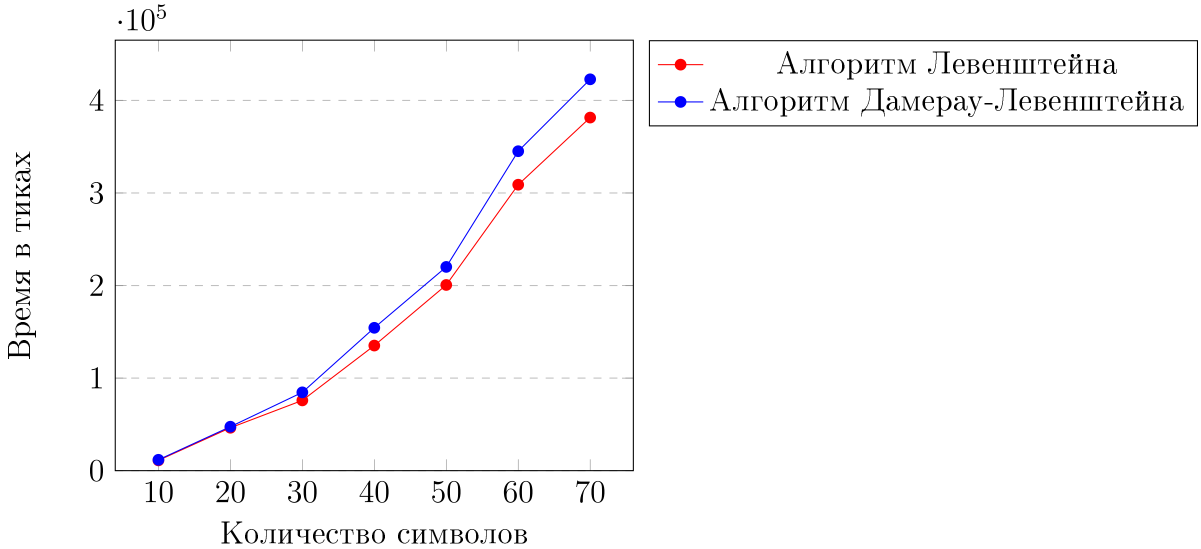

Рис. 4.2: График времени работы матричных реализаций алгоритмов Левенштейна и Дамерау-Левенштейна

4.3 Сравнение работы реализаций алгоритма Дамерау-Левенштейна

Для сравнения времени работы матричной и рекурсивной реализаций алгоритма Дамерау-Левенштейна были использованы строки длиной от 1 до 10 с шагом 1. Эксперимент для более точного результата повторялся 100 раз. Итоговый результат рассчитывался как средний из полученных результатов. Результаты измерений показаны в таблице 4.2 и на рисунках 4.3 и 4.4.

Время выполнения рекурсивной реализации алгоритма резко возрастает с увеличением длины слов: так при длине слова 5 рекурсивная выполняется в 15 раз дольше, чем матричная, а при длине слова 10 - приблизительно в 40000 раз. Рекурсивная реализация выигрывает по времени только при длине слов, равной 1 (в 2 раза), но это тривиальный случай. Можно сделать вывод о том, что матричная реализация алгоритма значительно эффективнее рекурсивной при любой длине слова.

Таблица 4.2: Время работы реализаций алгоритма Дамерау-Левенштейна в тактах процессора

Длина слова	Матричная реализация	Рекурсивная реализация
1	865	302
2	1184	1518
3	2097	7709
4	2250	35688
5	3575	199943
6	5778	1017271
7	7724	5901604
8	13897	34280680
9	16265	182931865
10	23011	1006037133

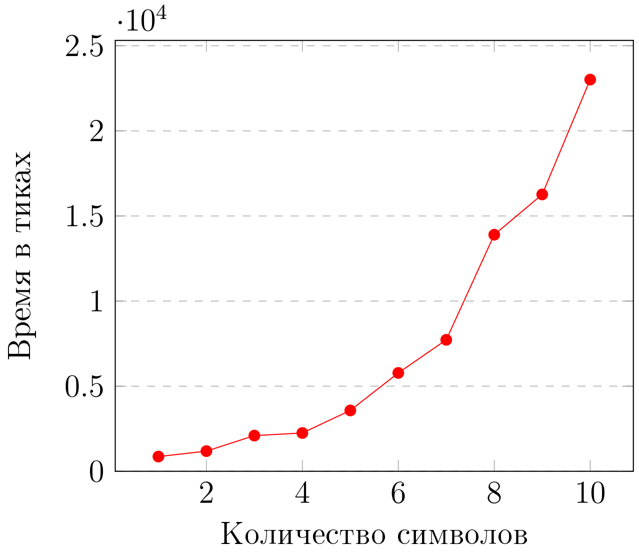

Рис. 4.3: График времени работы матричной реализации алгоритма Дамерау-Левенштейна

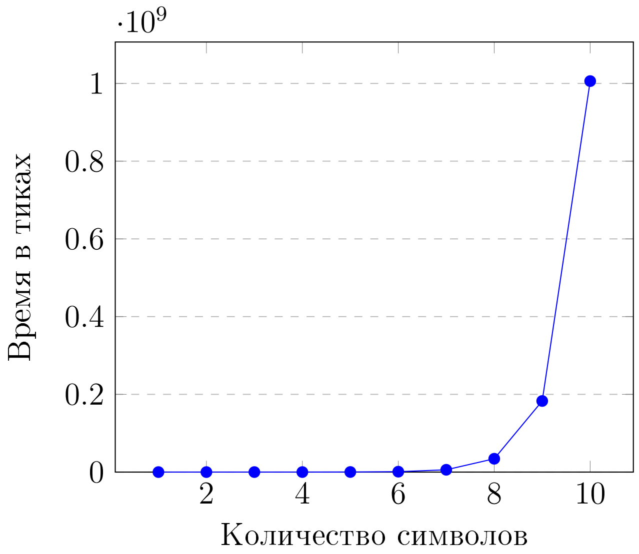

Рис. 4.4: График времени работы рекурсивной реализации алгоритма Дамерау-Левенштейна

Заключение

В ходе лабораторной работы были изучены и реализованы алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Для этого были реализованы три различные реализации алгоритмов с применением навыка динамического программирования для матричных.

Экспериментально потверждена эффективность матричных реализаций над рекурсивной: при длине слов выше 10 символов применение рекурсивной реализации алгоритма Дамерау-Левенштейна является нецелесообразной, т. к. проигрывает по памяти и по времени матричных в несколько порядков. Также было экспериментально установлено, что применение матричной реализации Дамерау-Левенштейна допустимо, так как данный алгоритм уступает по времени алгоритму Левенштейна лишь на 10%, но при это при решении определённых задач может давать меньший результат.

Литература

[1] Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. В. И. Левенштейн.

[2] A technique for computer detection and correction of spelling errors. Damerau Fred J.

[3] Indexing methods for approximate dictionary searching. Journal of Experimental Algorithmics, 2011. L. M. Boytsov

[4] https://cppreference.com/ [Электронный ресурс]