Государственное образовательное учреждение высшего профессионального образования
“Московский государственный технический университет имени Н.Э.Баумана”

Дисциплина: Анализ алгоритмов

Лабораторная работа № 1

ТЕМА

Студент группы ИУ7-54,
Лозовский Алексей

2019 г.

Содержание

Введение

На сегодняшний день трудно найти человека, ни разу не пользовавшегося интернетом. Нередко при наборе запроса можно допустить ошибку, однако поисковая система все равно корректно распознает введенный текст. Как ей это удается? Для решения таких задач используются алгоритмы нахождения редакционного расстояния или расстояния Левенштейна.

Редакционное расстояние между строками S1 и S2 или расстояние Левенштейна - минимальное количество редакторских операций (вставка, замена, удаление, совпадение), необходимых для преобразования одной строки в другую.

На сегодняшний день эти алгоритмы получили широкое применение в теории информации и компьютерной лингвистике. Кроме того, они активно применяются:

- для исправления ошибок в тексте, как описано в статье [1];
- для работы в сфере биоинформатики, в соответствии с информацией из [2] и [3].

1 Аналитическая часть

1.1 Описание алгоритмов

Расстояние Левенштейна — метрика, позволяющая определить «схожесть» двух строк — минимальное количество операций вставки, удаления и замены одного символа, необходимых для превращения одной строки в другую.

Расстояние Левенштейна между двумя строками a и b определяется функцией $d_{a,b}(|a|, |b|)$ как (1):

$$D_{a,b}(i,j) = \begin{cases} 0, & \text{i} = 0, \text{j} = 0 \\ i, & \text{j} = 0, \text{i} > 0 \\ j, & \text{i} = 0, \text{j} > 0 \\ min \begin{cases} D_{a,b}(i-1,j) + 1 \\ D_{a,b}(i,j-1) + 1 \\ D_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{i} > 0, \text{j} > 0 \end{cases} \tag{1}$$

где $1_{(a_i \neq b_j)}$ это индикаторная функция, равная нулю при $a_i = b_j$ и 1 в противном случае.

Одной из модификаций расстояния Левенштейна является *расстояние Дамерау-Левенштейна*, в ней к операциям вставки, удаления и замены символов, определённых в расстоянии Левенштейна добавляется операция транспозиции (перестановки) символов.

Таким образом, *Расстояние Дамерау — Левенштейна* между двумя строками a и b определяется функцией $d_{a,b}(|a|, |b|)$ как (2):

$$D_{a,b}(i,j) = \begin{cases} max(i,j) & \text{if min(i, j)} = 0, \\ min \begin{cases} D_{a,b}(i-1,j) + 1 \\ D_{a,b}(i,j-1) + 1 \\ D_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \\ D_{a,b}(i-2,j-2) + 1 \end{cases} & \text{if i, j} > 1 \text{ and } a_{i-1} = b_j \text{ and } a_i = b_{j-1} \\ min \begin{cases} D_{a,b}(i-1,j) + 1 \\ D_{a,b}(i,j-1) + 1 \\ D_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} & \text{otherwise} \end{cases} \tag{2}$$

где $1_{(a_i \neq b_j)}$ это индикаторная функция, равная нулю при $a_i = b_j$ и 1 в противном случае.

Данные алгоритмы могут быть реализованы несколькими методами: *матричным, рекурсивным и стековым.* Расстояние Левенштейна дало начало таким алгоритмам, как алгоритм Хиршберга и др.

1.2 Задание на выполнение лабораторной работы

- Реализовать матрично алгоритмы Левенштейна и Дамерау-Левенштейна, алгоритм Дамерау-Левенштейна рекурсивно;
- Алгоритмы сравнить по эффективности;
- Программа должна поддерживать два режима: пользовательский и экспериментальный;
- Протестировать работоспособность программы.

2 Конструкторская часть

2.1 Структура кода и представление данных

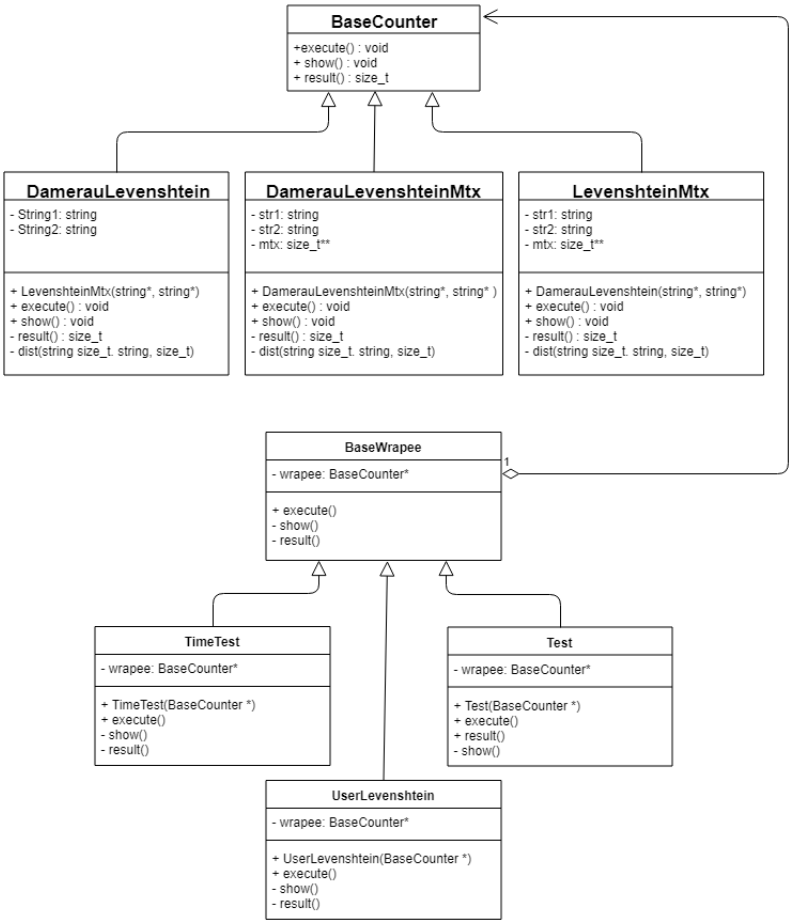

Рис. 1: Uml-схема

Каждый алгоритм оформлен в отдельном классе, унаследованном от базового - *BaseCounter*. Каждый из классов хранит в себе обрабатываемые строки, классы *DamerauLevenshteinMtx* и *LevenshteinMtx* также содержат поле матрицы, которая заполняется во время выполнения алгоритма.

Для режимов работы программы, пользовательского и экспериментального, используются TimeTest, Test и UserLevenshtein, которые представляют собой обертки, расширяющие функционал основных классов.

2.2 Разработка алгоритмов

2.2.1 Рекурсивный алгоритм Дамерау-Левенштейна

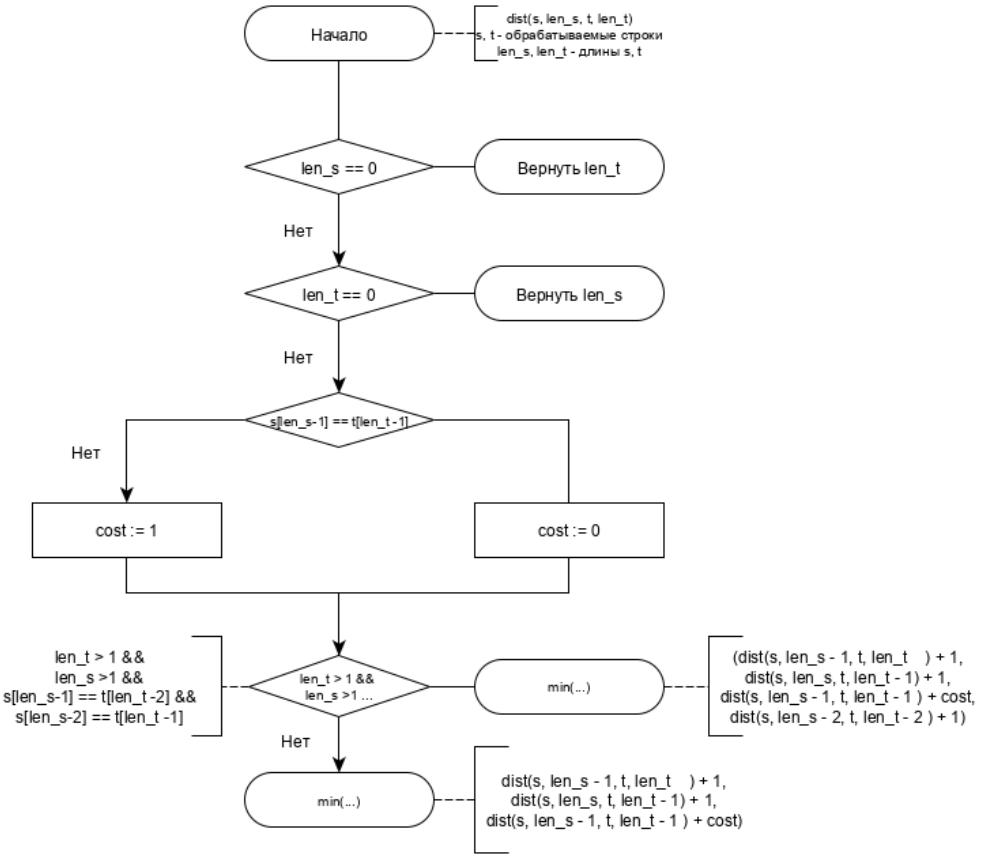

Рис. 2: Рекурсивный алгоритм

Алгоритм 1 Рекурсивный алгоритм Дамерау-Левенштейна $dist(s, t)$

$cost \leftarrow 0$
if $|s| = 0$ **then**
    return $|t|$;
if $|t| = 0$ **then**
    return $|s|$;
if $s[|s|] = t[|t|]$ **then**
    $cost \leftarrow 0$
else
    $cost \leftarrow 1$
return $min \begin{cases} dist(s[1 \dots |s| - 1], t) + 1 \\ dist(s, t[1 \dots |t| - 1]) + 1 \\ dist(s[1 \dots |s| - 1], t[1 \dots |t| - 1]|]) + cost \end{cases}$

2.2.2 Матричный алгоритм Дамерау-Левенштейна

Алгоритм 2 Матричный алгоритм Дамерау-Левенштейна $dist(s, t)$

$mtx[0 \dots m, 0 \dots n] \leftarrow 0$
for от $i = 0$ **из** [0 ... m] **do**
    $mtx[0, i] \leftarrow i$
for от $j = 0$ **из** [0 ... n] **do**
    $mtx[j, 0] \leftarrow j$

for от $i = 0$ **из** [0 ... m] **do**
    for от $j = 0$ **из** [0 ... n] **do**
        if $s[i] = t[j]$ **then**
            $mtx[i, j] = mtx[i - 1][j - 1]$
        else
            $mtx[i, j] = min(mtx[i - 1, j - 1] + 1, mtx[i - 1, j] + 1, mtx[i, j - 1] + 1)$

        if $i > 1 j > 1 s[i - 1] == t[j - 2] s[i - 2] == t[j - 1]$ **then**
            $mtx[i, j] = min(mtx[i, j], mtx[i - 2, j - 2] + 1)$
return $mtx[m, n]$

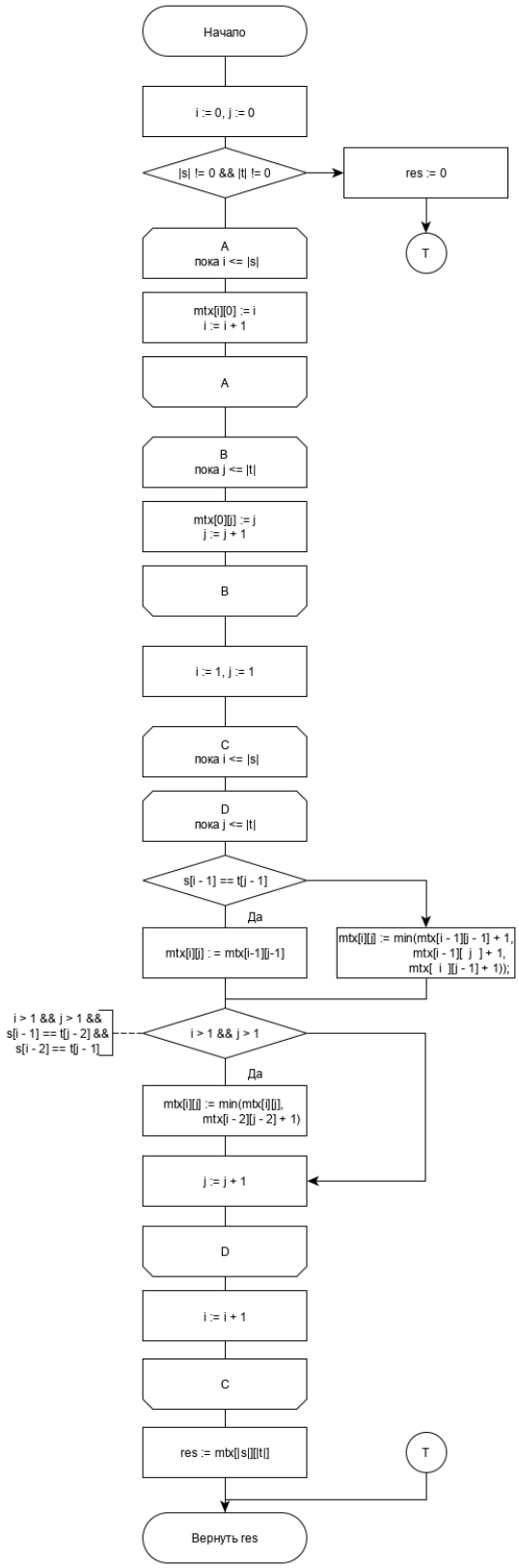

Рис. 3: Дамерау-Левенштейн матричная реализация

2.2.3 Матричный алгоритм Левенштейна

Алгоритм 3 Матричный алгоритм Левенштейна $dist(s, t)$

$mtx[0 \dots m, 0 \dots n] \leftarrow 0$
for от $i = 0$ **из** $[0 \dots \mathrm{m}]$ **do**
    $mtx[0, i] \leftarrow i$
for от $j = 0$ **из** $[0 \dots \mathrm{n}]$ **do**
    $mtx[j, 0] \leftarrow j$

for от $i = 0$ **из** $[0 \dots \mathrm{m}]$ **do**
    for от $j = 0$ **из** $[0 \dots \mathrm{n}]$ **do**
        if $s[i] = t[j]$ **then**
            $mtx[i, j] = mtx[i - 1][j - 1]$
        else
            $mtx[i][j] = min(mtx[i - 1][j - 1] + 1, mtx[i - 1][j] + 1, mtx[i][j - 1] + 1)$

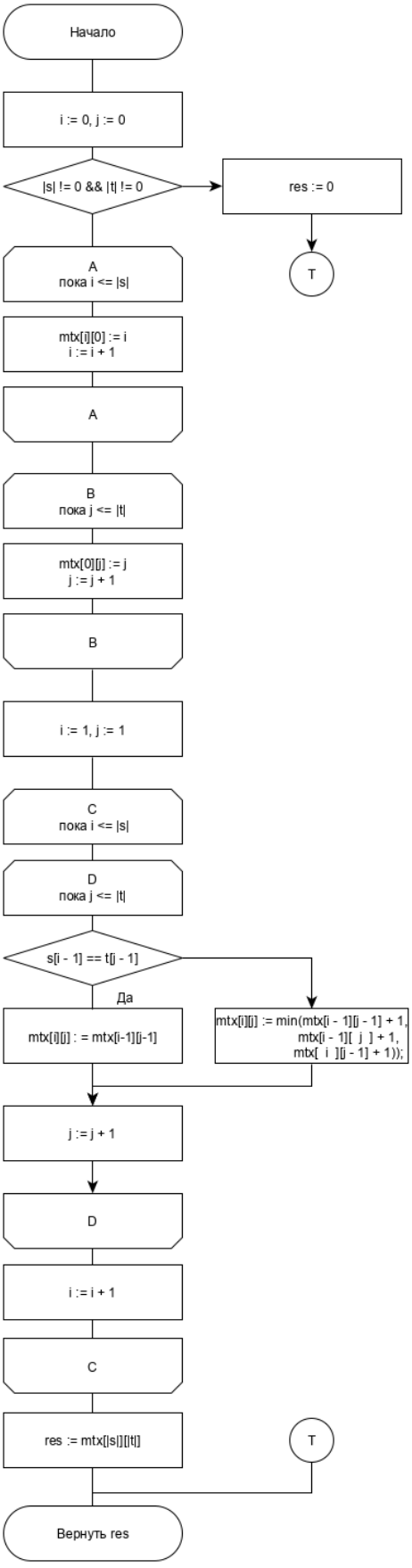

Рис. 4: Левенштейн матричная реализация

2.3 Выводы по конструкторскому разделу

Таким образом, была разработана общая структура программы и каждого ее режима. Были представлены схемы всех вычислительных алгоритмов. Каждый из них был описан на псевдокоде.

3 Технологическая часть

3.1 Требования к программному обеспечению

Программа должна поддерживать два режима - пользовательский и экспериментальный. В пользовательском режиме программа должна обеспечивать:

- ввод с клавиатуры;
- корректное завершение работы (без аварийных ситуаций).

3.2 Средства реализации

Поскольку структура программы поддерживает объектно-ориентированный принцип программирования, для ее разработки из языков, поддерживающих данный подход, был выбран язык C++, так как ранее уже имелся опыт работы с ним.

3.3 Листинг кода

Листинг 1: Рекурсивный алгоритм Дамерау-Левенштейна

```
size_t damerau_levenshtein::dist(std::string& s, size_t len_s, std::string& t,
   size_t len_t)
{
 if (len_s == 0)
  return len_t;

 if (len_t == 0)
  return len_s;

 size_t cost = (s[len_s - 1] == t[len_t - 1]) ? 0 : 1;

 if (len_s > 1 && len_t > 1 && s[len_s-1] == t[len_t-2] && s[len_s-2] ==
    t[len_t-1])
  return std::min({
   dist(s, len_s - 1, t, len_t ) + 1,
   dist(s, len_s, t, len_t - 1) + 1,
   dist(s, len_s - 1, t, len_t - 1 ) + cost,
   dist(s, len_s - 2, t, len_t - 2 ) + 1
 });

 else
  return std::min({
   dist(s, len_s - 1, t, len_t ) + 1,
   dist(s, len_s, t, len_t - 1) + 1,
   dist(s, len_s - 1, t, len_t - 1 ) + cost
 });
};
```

Листинг 2: Матричный алгоритм Дамерау-Левенштейна

```
size_t damerau_levenshtein_mtr::dist(std::string& s, size_t len_s, std::string&
   t, size_t len_t)
{
 for (size_t i=0; i<=len_str1; ++i)
  mtx[i][0] = i;

 for (size_t j=0; j<=len_str2; ++j)
  mtx[0][j] = j;

 if ((len_s == 0) and (len_t == 0))
  res = 0;
 else
 {
  for(size_t i = 1; i <= len_s; ++i)
  {
   for(size_t j = 1; j <= len_t; ++j)
   {
    if (s[i - 1] == t[j - 1])
     mtx[i][j] = mtx[i - 1][j - 1];
    else
     mtx[i][j] = min(mtx[i - 1][j - 1] + 1,
                   min(mtx[i - 1][ j ] + 1,
                       mtx[ i ][j - 1] + 1));

    if (i > 1 && j > 1 && s[i - 1] == t[j - 2] && s[i - 2] == t[j - 1])
     mtx[i][j] = min(mtx[i][j], mtx[i - 2][j - 2] + 1);
   }
  }
  res = mtx[len_s][len_t];
 }
 return res;
};
```

Листинг 3: Матричный алгоритм Левенштейна

```
size_t damerau_levenshtein_mtr::dist(std::string& s, size_t len_s, std::string&
   t, size_t len_t)
{
 for (size_t i=0; i<=len_str1; ++i)
 mtx[i][0] = i;

 for (size_t j=0; j<=len_str2; ++j)
 mtx[0][j] = j;

 if ((len_s == 0) and (len_t == 0))
 res = 0;
 else
 {
  for(size_t i = 1; i <= len_s; ++i)
  {
   for(size_t j = 1; j <= len_t; ++j)
   {
    if (s[i - 1] == t[j - 1])
```

```
    mtx[i][j] = mtx[i - 1][j - 1];
   else
    mtx[i][j] = min(mtx[i - 1][j - 1] + 1,
                  min(mtx[i - 1][ j ] + 1,
                                mtx[ i ][j - 1] + 1));

  }
 }
 res = mtx[len_s][len_t];
}
return res;
};
```

Для реализации тестов и пользовательского режима были реализованы классы-обертки, расширающие функционал основных классов. **Основные классы:** *damerau_ levenshtein, damerau_ lev levenstein_ mtx*
Классы обертки: *user_ levenshtein, TimeTest, Test*

Листинг 4: Класс для пользовательского режима

```
class UserLevenshtein : public base_counter
{
 public:
  UserLevenshtein(base_counter *);

  void execute() override;
  size_t result() override;

 private:
  base_counter *wrapee;

  void show() override;
};

UserLevenshtein::UserLevenshtein(base_counter *base)
{
 wrapee = base;
}

void UserLevenshtein::execute()
{
 wrapee->execute();
 show();
}


void UserLevenshtein::show()
{
 wrapee->show();
}
```

Листинг 5: Класс для режима тестирования

```
class Test : base_counter
{
 public:
  Test(base_counter *, size_t);

  void execute() override;

 private:
  base_counter *wrapee;
  size_t exp_res;

  void show() override;
  size_t result() override;

};

Test::Test(base_counter *base, size_t res)
{
 wrapee = base;
 exp_res = res;
}

void Test::execute()
{
 wrapee->execute();
 show();
}

size_t Test::result()
{
 return wrapee->result();
}

void Test::show()
{
 std::cout << "Test string1: " << wrapee->str1 << std::endl;
 std::cout << "Test string2: " << wrapee->str2 << std::endl;

 std::cout << "Expected result: " << exp_res << std::endl;
 std::cout << "Real result: " << wrapee->result() << std::endl;

 std::string message = wrapee->result() == exp_res ? "Complited." : "Failed.";

 std::cout << "Test: " << message << "\n" << std::endl;
}
```

3.4 Выводы по конструкторскому разделу

Был выбран язык программамирования для разработки программы, приведены листинги кода. Описаны требования к ПО.

4 Экспериментальная часть

4.1 Примеры работы

Входные данные:

Input the 1st string: **кот**
Input the 2nd string: **скат**

Выходные данные:

---

0 1 2 3
1 1 2 3
2 1 2 3
3 2 2 3
4 3 3 2
Levenshtein matrix result = 2
0 1 2 3
1 1 2 3
2 1 2 3
3 2 2 3
4 3 3 2
Damerau-Levenshtein matrix result = 2
Damerau-levenshtein recursion result = 2

Входные данные:

---

1. User-mode;
2. Func testing-mode;
3. Time testing-mode;
0. Exit;
Choose number: 5

Выходные данные:

---

incorrect input!

4.2 Постановка эксперимента

Для исследования работы реализаций алгоритмов нахождения редакционного расстояния будем подавать:

- Для матричных реализаций - строки длиной от 50 до 200;
- Для рекурсивной - равные и произвольные строки длиной до 13 символов;
- Для проверки корректности работы алгоритмов были подготовлены отдельные тесты. content

4.2.1 Тестирование работы функций

Рекурсивный алгоритм ДамерауЛевенштейна

Строка 1	Строка 2	Результат	Ожидаемый результат	Тест
кот	скат	2	2	Complete
seatback	backseat	8	8	Complete
kot	null	3	3	Complete
null	kot	3	3	Complete
null	null	0	0	Complete
matrial	martial	1	1	Complete

Матричный алгоритм ДамерауЛевенштейна

Строка 1	Строка 2	Результат	Ожидаемый результат	Тест
кот	скат	2	2	Complete
seatback	backseat	8	8	Complete
kot	null	3	3	Complete
null	kot	3	3	Complete
null	null	0	0	Complete
matrial	martial	1	1	Complete

Матричный алгоритм Левенштейна

Строка 1	Строка 2	Результат	Ожидаемый результат	Тест
кот	скат	2	2	Complete
seatback	backseat	8	8	Complete
kot	null	3	3	Complete
null	kot	3	3	Complete
null	null	0	0	Complete
matrial	martial	2	2	Complete

4.2.2 Тестирование времени работы функций

Строка 1	Строка 2	МДЛ	МЛ	РДЛ
200	200	0.0011	0.0008	
100	100	0.0002	0.0001	
50	50	0.0001	0.0001	
10	10	0	0	0.1634
8	8	0	0	0.0054
5	5	0	0	0.0001
50	2	0	0	0.0001
50	4	0	0	0.0851

4.2.3 Память

Матричные алгоритмы

Для матричных алгоритмов используются:

- строка1 : string ;
- строка2 : string ;
- длина строки1 : size_t ;
- длина строки2 : size_t ;
- матрица : size_t** ;
- результат : size_t .

Таким образом, используемую память помжно вычислить по формуле:

```
auto mem = 3 * sizeof(size_t) + str1.size() + str2.size() + len_str1 *
   len_str2 * sizeof(size_t) + sizeof(size_t*)*len_str1;
```

Рассмотрим на примере.
Для высчитывания расстояния Левенштейна и Дамерау-Левенштейна между строками ‘kot‘ и ‘skat‘ потребуется 151 байт памяти.

Рекурсивный алгоритм

Для рекурсивного алгоритма можно получить количество вызовов функции. Для этого заведем счетчик $n = 0$, при каждом вызове будем увеличивать его на 1.

Таким образом, при тестировании, было определено, что для обработки строк ‘kot‘ и ‘skat‘ функция была вызвана 193 раза.
Для данного алгоритма используются:

- строка1 : string ;
- строка2 : string ;

- длина строки1 : size_t ;
- длина строки2 : size_t ;
- результат : size_t ;

Вычислим используюемую память

```
auto mem = 3 * sizeof(size_t) + str1.size() + str2.size();
```

Для работы со строками ‘kot‘ и ‘skat‘ потребуется 31 байт памяти.

4.3 Сравнительный анализ на материале экспериментальных данных

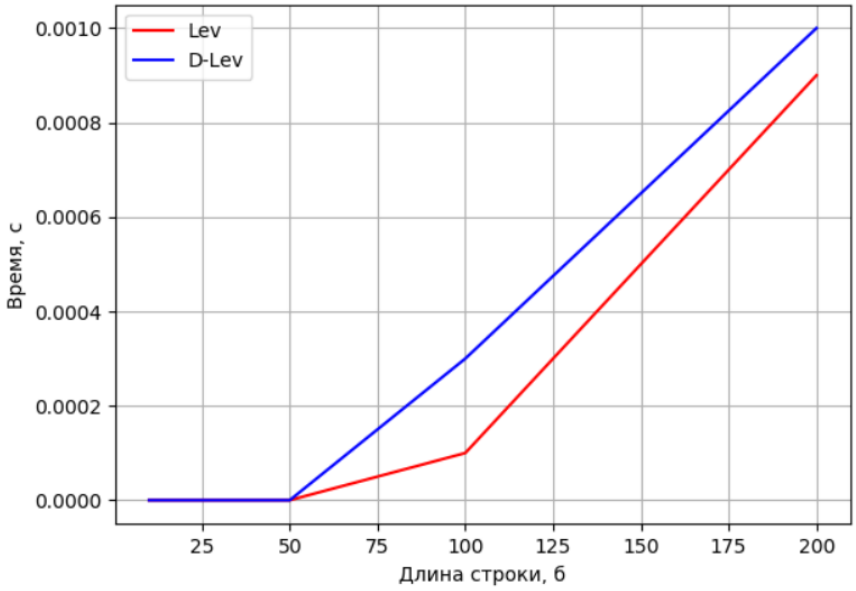

Рис. 5: Сравнение матричных реализаций алгоритмов

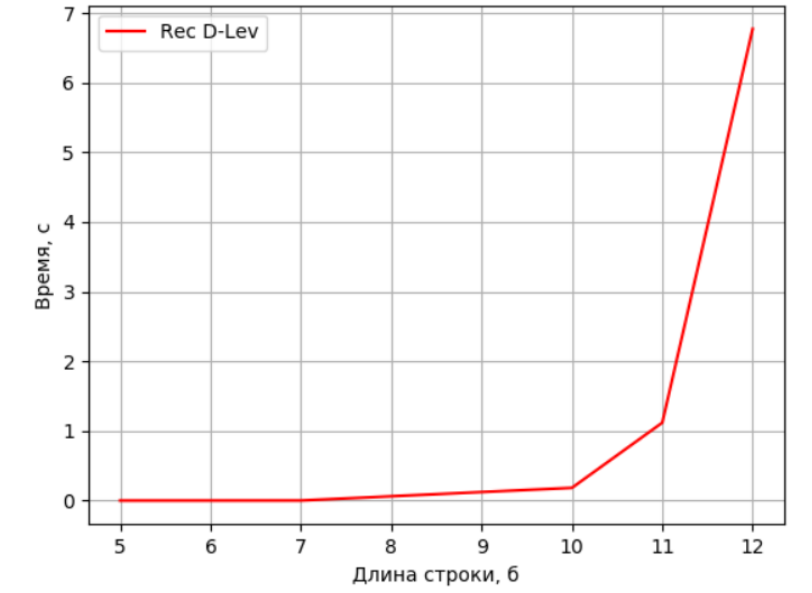

Рис. 6: Работа рекурсивного алгоритма

На рис. 5 показана работа матричных реализаций алгоритма поиска редакционного расстояния. Исходя из значений графика, можно сделать вывод, что алгоритм Левенштейна начинает работать быстрее при длине строк, большей или равной 50.

На рис. 6 отображена работа рекурсивного алгоритма Дамерау-Левенштейна. Алгоритм обрабатывает строки, длиной до 10 символов меньше, чем за секунду, но замедляется при обработке данных, большего размера.

4.4 Выводы по экспериментальной части

На основе полученных в результате тестирования данных, можно сделать следующие выводы:
Матричный алгоритм Левенштейна работает быстрее, чем его модификация. Чем больше размер обрабатываемых строк, тем сильнее отличается время работы данных алгоритмов Так, при обработке двух строк, длиной 50 символов алгоритмы показали одинаковые результаты - 0.0001с. При дальшейшем тестировании на строках длиной в 100 и 200 символов, было получено - 0.0002с, 0.0001с и 0.0008с, 0.0011с соответственно.
Рекурсивный алгоритм показал худшие результаты при работе со строками одинакового размера. При подаче данных длиной до 6 символов, программа время совпадает со временем матричных реализаций. При обработке строк длиной от 7байт алгоритм начинает работать медленнее. При длине в 7 символов программа отработала за 0.0001с, для 10 - 0.1646с, после 10 символов программа замедляется - для 11 символьных строк - 0.963с, для 12 - 7.4164с. При работе со строками большего размера, время ожидания не определено.
Рекурсивный алгоритм показал неплохие результаты при работе с данными разных величин - при подаче одной большой строки (до 50 символов) и второй до 4, программа работает за 0.0851с Таким образом, для обработки равных строк, длиной до 200с быстрее всего работает алгоритм Левенштейна - 0.0008с, Самые плохие результаты показал рекурсивный алгоритм, при строках 5 символов, матричные реализации показали результат 0, Рекурсивная - 0.0001с

Заключение

Таким образом, в ходе лабораторной работы была написана программа с двумя режимами пользовательским и экспериментальным, в которой были реализованы алгоритм Левенштейна и его модификации. В результате тестирования программы были сделаны выводы об эффективности данных алгоритмов. Было определено, что матричные реализации эффективнее по времени, чем рекурсивная, при обработке строк малой длины(до 10 символов) первая группа алгоритмов отработала за 0.0000с, в то время как рекурсивный аналог обрабатал данные строки за 0.1646с. При обработке строк большего размера, время работы рекурсивного алгоритма установить не удалось. Также рограмма была протестирована на работоспособность - при подаче различных данных приложение работает корректно.

Список литературы

[1] Damareau , F., J., "A technique for computer detection and correction of spelling errors"

[2] US National Library of Medicine National Institutes of Health, "Secure approximation of edit distance on genomic data"

[3] The Institute of Electronics, "Tree Edit Distance Problems: Algorithms and Applications to Bioinformatics"

[4] Ovicic, V., "Constrained edit distance algorithm and its application in Library Information Systems"