Отчет по лабораторной работе №1
по курсу "Анализ алгоритмов"
по теме "Расстояние Левенштейна"

Студент: Барсуков Н.М. ИУ7-56
Преподаватель: Волкова Л.Л., Строганов Ю.В.

Содержание

Введение

Расстояние Левенштейна (также редакционное расстояние или дистанция редактирования) между двумя последовательностями символов. В теории информации и компьютерной лингвистике — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна и его обобщения активно применяются [4]:

1. для исправления ошибок в слове (в поисковых системах, базы данных, при вводе текста, при автоматическом распознавании отсканированного текста и речи);
2. в биоинформатике для сравнения генов, хромосом, белков.

1 Аналитическая часть

Цель: Изучить, реализовать и сравнить алгоритмы вычисления редакционного расстояния между 2 строками

Для этого необходимо выполнить следующие пункты:

1. Иследовать алгоритмы вычисления редакционного расстояния;
2. Реализовать следующие алгоритмы:
   (a) Итеративный;
   (b) Рекурсивный;
   (c) Модифицированный итеративный.
3. Сделать замеры выбранных алгоритмов;
4. Сравнить результаты замеров;
5. Сделать вывод.

В нашей лабораторной работе будет рассматриваться 2 способа реализации данного алгоритма.

1. Итеративный;
2. Рекурсивный.

Все они работает по следующему принципу. Допустим,что существует две строки S1 и S2 над некоторым алфавитом. Длина одной из них - M, второй - N. Для нахождения расстояния Левенштейна между ними D(S1, S2) можно применить следующую формулу (D(S1, S2) == D(M, N))[4] (1 2):

$$D(i,j) = \begin{cases} max(i,j)\ if\ min(i,j) == 0 \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+(S1[i] <> S2[j]) \end{cases} \end{cases} \tag{1}$$

$$D(i,j) = \begin{cases} max(i,j)\ if\ min(i,j) == 0 \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+(S1[i] <> S2[j]) \\ D(i-2,j-2)+1 \end{cases} \quad if\ i,j > 1 \\ min \begin{cases} D(i,j-1)+1 \\ D(i-1,j)+1 \\ D(i-1,j-1)+(S1[i] <> S2[j]) \end{cases} \end{cases} \tag{2}$$

Так же вводится понятие стоимости операций. По умолчанию предполагается, что каждая операция, кроме операции сравнения символом, стоят по 1. Введем собственную стоимость для нашего случая:

1. Вставка в S1 (I) - 1;
2. Удаление из S1 (D) - 1;
3. Замена символа в S1 (R) - 1;
4. Обмен местами 2 соседних символов (S) - 1;
5. Совпадение символов в S1 и S2 - 0.

1.0.1 Вывод

В данном разделе были рассмотрены алгоритмы поиска редакционного расстояния Левенштейн и Левенштейн - Дамерау, который является модифицированной версией первого и вводит дополнительную операцию перестановки двух ближних символов

Мы будем ализовывать 2

2 Конструкторский раздел

В данном разделе буду приведены блок схемы и описана структура программы

2.1 Алгоритм

В данном подразделе буду представлены схемы итеративного(1), рекурсивного(2), модифицированного алгоритма(3). Как вы можете заметить схема отличается только дополнительной проверкой и введением дополнительной операции.

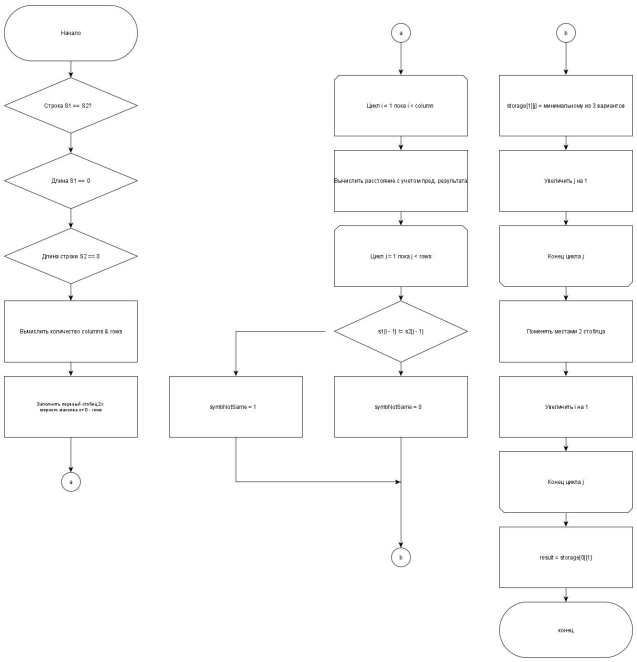


Рис. 1: Схема итеративного алгоритма

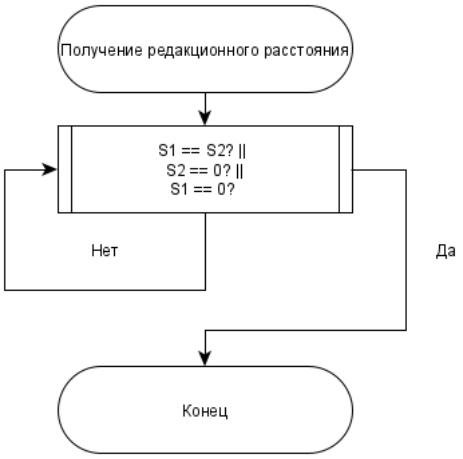


Рис. 2: Блок схема рекурсивного алгоритма

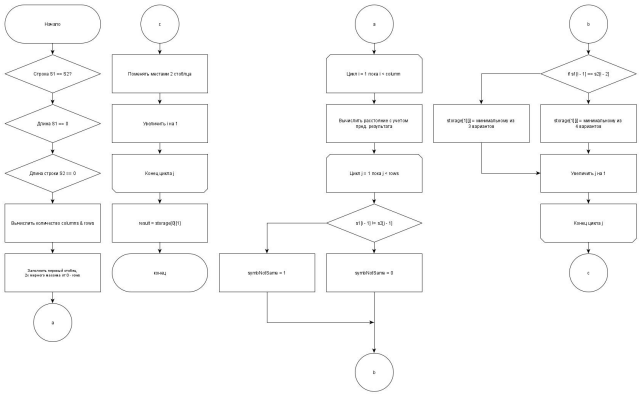


Рис. 3: Блок схема мод. алгоритма

2.2 Структура программы

Программа обладает следующей структурой:

1. приветственное окно в котором пользователся знакомят с программой и просят выбрать режим работы. Режимы следующие:
   (a) поиск редакционного расстояния на 2 строках введенных пользователем;
   (b) поиска редакционного расстояния на случайных строках длины указанной пользователем.

2.3 Вывод

В данном разделе были представлены схемы итеративного (матричного), рекурсивного и модифицированного Левенштейна и структура программы

3 Технологический раздел

В данном разделе будут описаны требования к программному продукту и инструментарий реализации. Приведен интерфейс программы и листинг кодов алгоритмов

3.1 Требования

Требования к данному программному продукту следующие:

1. программа должна корректно работать;
2. возвращать редакционное расстояние между 2 строк длинной до 10000 символов;
3. не должна обрабатывать некорректный вод, ответственность ложиться на пользователя

3.2 Выбор языка и среды разработки

Для решения данной поставленной задачи, мной был выбран язык c++ стандарта c11 по причине использования структурного подхода и его мультиплатформенности.

Так же мной используются среда разработки под названием Visual Studio 2019 по причине удобства отладки и функциональности. И так же желания изучить данную среду по лучше.

3.3 Интерфейс

Интерфейс представляет из себя простую консоль в котором пользователь взаимодействует с помощью ввода команд. Такой тип взаимодействия выбран, по причине простоты разработки и удобства тестирования программы (4, 5).

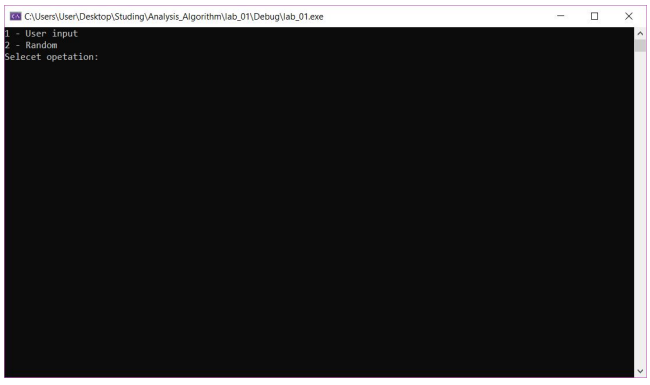


Рис. 4: Примера работы

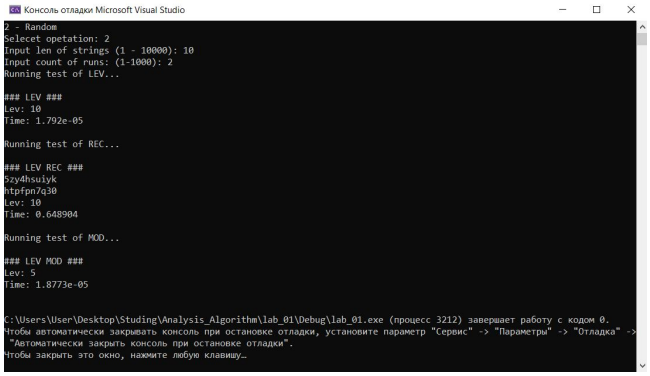


Рис. 5: Примера работы

3.4 Листинг

Ниже будут представлены листинги итеративного (1), рекурсивного (2), модифицированного (3)

Листинг 1: Итеративный алгоритм

```
1  int levenstain3(char* str1, char* str2) {
2
3      if (strcmp(str1, str2) == 0) {
4          return 0;
5      }
6
7      if (strlen(str1) == 0) {
8          return strlen(str2);
9      }
10
11     if (strlen(str2) == 0) {
12         return strlen(str1);
13     }
14
15     unsigned columns = strlen(str1) + 1;
16     unsigned rows = strlen(str2) + 1;
17
18     int* storage[2];
19     storage[0] = new int[rows];
20     storage[1] = new int[rows];
21
22     for (unsigned i = 0; i < rows; i++) {
23         storage[0][i] = i;
24     }
25
26     for (unsigned i = 1; i < columns; i++) {
27         storage[1][0] = storage[0][0] + 1;
28         for (unsigned j = 1; j < rows; j++) {
29             bool symbNotSame = false;
```

```
30              if (str1[i - 1] != str2[j - 1]) {
31                  symbNotSame = true;
32              }
33
34              storage[1][j] = min(min(
35                                  storage[0][j] + 1,
                                        storage[1][j - 1] +
                                        1),
36                                  storage[0][j - 1] +
                                        symbNotSame);
37          }
38          swap(storage[0], storage[1]);
39      }
40
41      int result = storage[0][rows - 1];
42
43      delete storage[0];
44      delete storage[1];
45
46      return result;
47  }
```

Листинг 2: Рекурсивный алгоритм

```
1   int levenstainRec3(char* str1, char* str2) {
2
3       if (strcmp(str1, str2) == 0) {
4           return 0;
5       }
6
7       if (strlen(str1) == 0) {
8           return strlen(str2);
9       }
10
11      if (strlen(str2) == 0) {
12          return strlen(str1);
13      }
14
15
16      bool symbNotSame = false;
17      if (*str1 != *str2) {
18          symbNotSame = true;
19      }
20
21      int result = min(min(levenstainRec3(str1 + 1, str2)
            + 1, levenstainRec3(str1, str2 + 1) + 1),
22                       levenstainRec3(str1 + 1, str2 + 1) +
                             symbNotSame);
23
```

```
24      return result;
25  }
```

Листинг 3: Левенштейн модифицированный

```
1   int levenstain4(char* str1, char* str2) {
2
3       unsigned lenstr1 = strlen(str1);
4       unsigned lenstr2 = strlen(str2);
5
6       if (lenstr1 < 2 || lenstr2 < 2)
7           return levenstain3(str1, str2);
8
9       unsigned columns = lenstr1 + 1;
10      unsigned rows = lenstr2 + 1;
11
12      int* storage[3];
13      for (int i = 0; i < 3; i++)
14          storage[i] = new int[rows];
15
16      for (unsigned i = 0; i < rows; i++) {
17          storage[0][i] = i;
18      }
19
20      storage[1][0] = 1;
21      for (unsigned i = 1; i < rows; i++) {
22          bool sumbNotSame = false;
23          if (str1[i - 1] != str2[0]) {
24              sumbNotSame = true;
25          }
26          storage[1][i] = min(min(storage[1][i - 1] + 1,
                storage[0][i] + 1), storage[0][i - 1] +
                sumbNotSame);
27      }
28
29      for (unsigned i = 2; i < columns; i++) {
30          storage[2][0] = storage[1][0] + 1;
31          bool symbNotSame = false;
32
33          if (str1[1] != str2[i - 1]) {
34              symbNotSame = true;
35          }
36
37          storage[2][1] = min(min(storage[2][0] + 1,
                storage[1][1] + 1), storage[1][0] +
                symbNotSame);
38
39          for (unsigned j = 2; j < rows; j++) {
40              symbNotSame = false;
```

```
41                 if (str2[j - 1] != str1[i - 1]) {
42                     symbNotSame = true;
43                 }
44
45                 int tmp = min(min(min(
46                     storage[1][j] + 1, storage[2][j - 1] +
                           1),
47                     storage[1][j - 1] + symbNotSame),
48                     storage[0][j - 2] + 1);
49
50                 storage[2][j] = tmp;
51             }
52
53             swap(storage[0], storage[1]);
54             swap(storage[1], storage[2]);
55         }
56
57         int result = storage[1][rows - 1];
58
59         delete storage[0];
60         delete storage[1];
61         delete storage[2];
62
63         return result;
64     }
```

3.5 Вывод

В данном разделе вы ввели требования по программе, выбрали среду и язык разработки, так же определились с будущим интерфейсом нашей программы

4 Исследовательский раздел

В данном разделе представлена конфигурация компьютера на котором проводилось тестирование. Результаты замером и сравнение алгоритмов

4.0.1 Железо

1. Компьютер:
   (a) Тип компьютера Компьютер с ACPI на базе x64;
   (b) Операционная система Microsoft Windows 10 Pro.
2. Системная плата:
   (a) тип ЦП DualCore Intel Core i5-6200U, 2700 MHz (27 x 100);
   (b) системная плата HP 8079;
   (c) чипсет системной платы Intel Sunrise Point-LP, Intel Skylake-U;
   (d) системная память 8072 МБ (DDR4 SDRAM).

4.1 Сравнение

4.1.1 Общие результаты замеров

Таблица 1(1) содержит в себе общие результаты замеров времени, необходимого для вычисления редакционного расстояния между 2 строками равной длины. Считается что слова полностью различны.

Условия замеров:

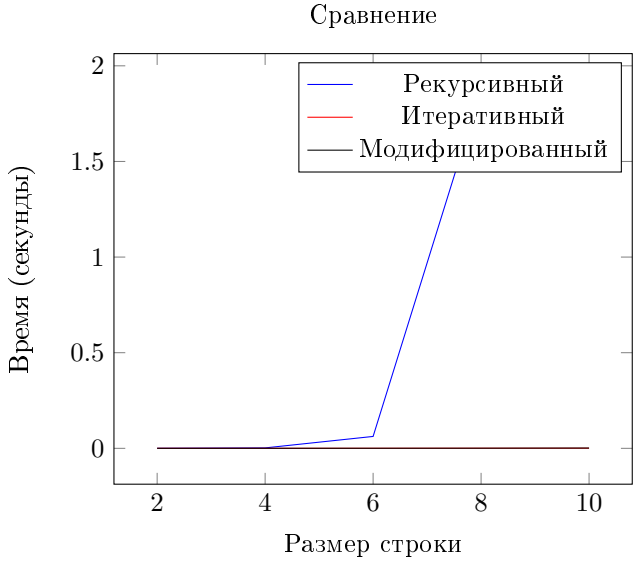
1. Время
   (a) Время измерений: секунды;
   (b) Берется суммарное время за все итерации
2. Количество итераций алгоритма: 100;
3. Символы в строках полностью различны

Таблица 1: Общие результаты замеров на 100 итераций

LEN	Рекурсивный	Итеративаный	Модифицированный
2	0.000105387	0.000169814	0.000147626
4	0.00224298	0.000243199	0.0003136
6	0.0621811	0.000329813	0.000431786
8	1.87619	0.000616959	0.000716373
10	56.2996	0.000702292	0.00151893

Таблица 2: Таблица сравнения времени

Длина слов	Сравнение с Итеративным	Сравнение с Модифицированным
2	-6.442700000000001e-05	-4.22390000000000014e-05
4	0.001999781	0.0019293799
6	0.061851287	0.061749314
8	1.875573041	1.875473627
10	56.298897708	56.29808



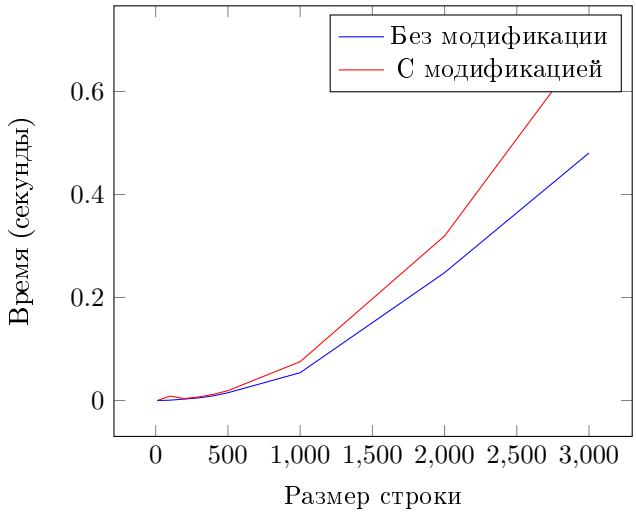
В таблице (2) находятся результаты сравнения рекурсивной с другими реализациями алгоритма. Как мы можем заметить на строках длины 2 рекурсивный справился быстрее на -6.442700000000001e-05 итеративного и на -4.22390000000000014e-05 быстрее модифицированного, но начиная с длинны строк равным 4 и больше он начает проигрывать по времени. В следствии этого дальнейшее детальное сравнение рекурсивного и другими проводитсья не будет.

4.1.2 Детальное сравнение

Сравним детально время работы классического Левенштейна и модифицированого (с операцией перестановки). На каждый метод берется среднее время 1000 прогонов.

L	Итеративаный	Модифицированный
10	6.322e-06	1.112e-05
100	0.000724276	0.00859501
200	0.00286479	0.00383968
300	0.00517626	0.00711481
400	0.0091726	0.0121067
500	0.0151016	0.0191977
1000	0.0540165	0.0753726
2000	0.248242	0.319514
3000	0.480696	0.696582

Сравнение



4.2 Вывод

В данном разделе были проведены замеры времени необходимого для вычисления редакционного расстояния трех различных алгоритмов зависимости от:

1. Длины слов;
2. От их редационного расстояния

Рекурсивный алгоритм при более простой реализации работает чрезвычайно долго. На пример: для строк длины 8 при суммарном времени 100 прогонов, он работает 1.87 дольше итеративного и модифицированного, что делает его использование нецелесообразным. Итеративный алгоритм и его модификация значительно превосходит его по эффективности.

Не смотря схожую ассимптику Левенштейна и Дамерау - Левенштейн, второй работает в среднем медленнее из более сложной внутренней реализации

Заключение

В ходе работы было проведено сравнение алгоритмов поиска расстояния Левенштейна (рекурсивной и итеративной реализации) и Дамерау-Левеншнейна (итеративной реализации). Были исследованы зависимости времени выполнения программ, реализующих данные алгоритмы, от искомого расстояния и от размеров строк для случаев с одинаковой длиной исходных строк и случая, когда одна из строк значительно меньше другой. В ходе исследования были сделаны следующие выводы:

1. Рекурсивная реализация алгоритма Левенштейна выполняется за приемлемое время лишь в случаях, когда размер одной из строк до 10-15 символов. Среднее время составляет 0.5 секунды, что в разы дольше модифицированного и тем более итеративного.

2. 2) Итеративные реализации алгоритмов поиска расстояний Дамерау-Левенштейна и Левенштейна имеют схожую ассимптитику, но алгоритм поиска расстояния Дамерау-Левенштейна из-за более сложной внутренней логики в среднем работает медленнее.

Список литературы

[1] Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход.-М.:Техносфера, 2009.

[2] Методика идентификации пассажира по установленным данным В. М . Черненький , Ю. Е . Гапанюк [Электронный ресурс]. – Режим доступа: http://www.engjournal.ru/articles/89/89.pdf, свободный – (07.10.2019)

[3] Библиография в LaTeX [Электронный ресурс]. – Режим доступа: https://habr.com/ru/post/114997/, свободный – (26.09.2019) https://habr.com/ru/post/114997/

[4] Нечеткий поиск, расстояние левенштейна алгоритм [Электронный ресурс]. – Режим доступа: https://steptosleep.ru/antananarivo-106/, свободный – (01.10.2019)

[]