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»

Лабораторная работа № 2
по дисциплине ”Анализ алгоритмов”

Тема Сравнение алгоритмов матричного умножения

Студент Калашников С.Д.

Группа ИУ7-53Б

Преподаватель Волкова Л.Л. Строганов Ю.В.

Москва, 2022

СОДЕРЖАНИЕ

Введение

Операции работы со строками являются важными компонентами в программирования. Часто возникает потребность в использовании строк при решении различных задач, в которых нужны алгоритмы сравнения строк, о которых и пойдет речь в данной работе. Одними из самых популярных алгоритмов в данной сфере являютя алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Расстояние Левенштейна — минимальное количество редакционных операций (вставка, удаление, замена символа), необходимых для преобразования одной строки в другую.

Если текст был набран с клавиатуры, то вместо расстояния Левенштейна чаще используют расстояние Дамерау-Левенштейна, в котором добавляется еще одно возможное действие — перестановка двух соседних символов.

Расстояния Левенштейна и Дамерау-Левенштейна применяются в таких сферах, как:

- компьютерная лингвистика (автозамена в посиковых запросах, текстовая редактура);
- биоинформатика (последовательности белков);
- нечеткий поиск записей в базах (борьба с мошенниками и опечатками).

Целью данной лабораторной работы является изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна.

Для достижения поставленной цели требуется решить ряд задач:

1) изучить алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;

2) разработать алгоритмы поиска этих расстояний;

3) реализовать каждый из данных алгоритмов;

4) провести замеры процессорного времени для каждой из реализаций алгоритмов;

5) расчитать затрачиваемую реализованными алгоритмами пиковую память;

6) выполнить анализ полученных результатов;

7) по итогам работы составить отчет.

1. Аналитическая часть

В данном разделе будут разобраны алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна (р. Л.) между двумя строками — метрика, позволяющая определить «схожесть» двух строк — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую (каждая операция имеет свою цену — штраф).

Редакционное предписание — последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену (и является расстоянием Левенштейна). Пусть S_1 и S_2 — две строки, длиной N и M соответственно. Введены следующие обозначения:

- I (англ. Insert) — вставка символа в произвольной позиции ($w(\lambda, b) = 1$);
- D (англ. Delete) — удаление символа в произвольной позиции ($w(\lambda, b) = 1$);
- R (англ. Replace) — замена символа на другой ($w(a, b) = 1,\ a \neq b$);
- M (англ. Match) — совпадение двух символов ($w(a, a) = 0$).

С учетом введенных обозначений, расстояние Левенштейна может быть подсчитано

по формуле

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

где функция $m(a,b)$ определена как:

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

1.1.1 Матричный алгоритм нахождения расстояния

Рекурсивный алгоритм вычисления расстояния Левенштейна может быть не эффективен при больших i и j, так как множество промежуточных значений $D(i,j)$ вычисляются не один раз, что сильно замедляет время выполнения программы.

В качестве структуры данных для хранения промежуточных значений можно использовать *матрицу*, имеющую размеры:

$$(length(S1)+1) \times ((length(S2)+1), \tag{1.3}$$

где $length(S)$ — длина строки S

Значение в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец заполнены нулями.

Всю таблицу (за исключением первого столбца и первой строки) заполняем в

соответствии с формулой

$$A[i][j] = min \begin{cases} A[i-1][j] + 1 \\ A[i][j-1] + 1 \\ A[i-1][j-1] + m(S1[i], S2[j])) \end{cases}, \tag{1.4}$$

где функция $m(S1[i], S2[j])$ определена как:

$$m(S1[i], S2[j]) = \begin{cases} 0, & \text{если } S1[i-1] = S2[j-1], \\ 1, & \text{иначе} \end{cases}. \tag{1.5}$$

Результат вычисления расстояния Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна (р. Д-Л.) между двумя строками, состоящими из конечного числа символов —– это минимальное число операций вставки, удаления, замены одного символа и транспозиции двух соседних символов, необходимых для перевода одной строки в другую.

Является модификацией расстояния Левенштейна — добавлена операции транспозиции, то есть перестановки, двух символов.

Расстояние Дамерау-Левенштейна может быть найдено по формуле

$$d_{a,b}(i,j) = \begin{cases} \max(i,j), & \text{если } \min(i,j) = 0, \\ \min\{ & \\ \quad d_{a,b}(i,j-1) + 1, & \\ \quad d_{a,b}(i-1,j) + 1, & \\ \quad d_{a,b}(i-1,j-1) + m(a[i],b[j]), & \text{иначе} \\ \quad \begin{cases} d_{a,b}(i-2,j-2) + 1, & \text{если } i,j > 1; \\ & a[i] = b[j-1]; \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases}, \tag{1.6}$$

Формула выводится по тем же соображениям, что и формула (1.1).

1.2.1 Рекурсивный алгоритм нахождения расстояния

Рекурсивный алгоритм вычисления расстояния Дамерау-Левенштейна реализует формулу 1.6

Минимальная цена преобразования — минимальное значение приведенных вариантов.

Если полагать, что *a', b'* — строки a и b без последнего символа соответственно, а *a'', b''* — строки a и b без двух последних символов, то цена преобразования из строки a в b выражается из элементов, представленных ниже:

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 на разные символы;

4) сумма цены преобразования из a'' в b'' и операции перестановки, предполагая, что длины a'' и b'' больше 1 и последние два символа a'', поменянные местами, совпадут с двумя последними символами b'';

5) цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

1.2.2 Матричный алгоритм нахождения расстояния

Рекурсивный алгоритм вычисления расстояния Дамерау-Левенштейна может быть не эффективен при больших i и j, так как множество промежуточных значений $D(i,j)$ вычисляются не один раз, что сильно замедляет время выполнения программы.

В качестве структуры данных для хранения промежуточных значений можно использовать матрицу, имеющую размеры:

$$(length(S1)+1) \times ((length(S2)+1), \tag{1.7}$$

где $length(S)$ — длина строки S

Значение в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец тривиальны.

Всю таблицу (за исключением первого столбца и первой строки) заполняем в соответствии с формулой

$$A[i][j] = min \begin{cases} A[i-1][j]+1 \\ A[i][j-1]+1 \\ A[i-1][j-1]+m(S1[i], S2[j])) \\ A[i-2][j-2]+1, \text{если } i>1,\ j>1 \text{ и} \\ \qquad S1[i-2]=S2[j-1],\ S2[i-1]=S2[j-2] \end{cases}, \tag{1.8}$$

где функция $m(S1[i], S2[j])$ определена как:

$$m(S1[i], S2[j]) = \begin{cases} 0, & \text{если } S1[i-1] = S2[j-1], \\ 1, & \text{иначе} \end{cases} \tag{1.9}$$

Результат вычисления расстояния Дамерау-Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.2.3 Рекурсивный алгоритм нахождения расстояния с использованием кеша

Чтобы уменьшить время работы рекурсивного алгоритма заполнения можно использовать кеш, который будет представлять собой матрицу.

Ускорение достигается за счет использования матрицы для предотвращения повторной обработки уже обработанных данных.

Если данные ещё не были обработаны, то результат работы рекурсивного алгоритма заносится в матрицу. В случае, если обработанные данные встречаются снова, то для них расстояние не находится и выполняется следующий шаг.

1.3 Вывод

В данном разделе были рассмотрены алгоритмы поиска расстояния Левенштейна и расстояния Дамерау-Левенштейна. В частности были приведены рекурентные формулы работы алгоритмов, объяснена разница между расстоянием Левенштейна и расстоянием Дамерау-Левенштейна.

2. Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги сортировок.

2.1 Средства реализации

В данной работе для реализации был выбран язык программирования $c++$. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы. Для визуализации результатов использовался язык $Python$.

Время работы реализаций алгоритмов было замерено с помощью функции *GetProcessTimes(...)* [1] из библиотеки $Windows.h$. Функция возвращает пользовательское процессорное временя типа float. Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

2.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- $AA2.cpp$ — файл, содержащий весь служебный код;
- $Matrix.cpp$ — файл, реализующий операции над матрицами;
- $algoc.cpp$ — файл, содержаший реализации алгритмов поиска расстояний Л. и Д-Л.;
- $TimeCompare.cpp$ — файл, производящий замеры времени;
- $GetCPUTime.cpp$ — файл, определяющий функцию замера времени;

- *Gen.cpp* — файл, генерирующий входную матрицу;
- *Matrix.hpp* — заголовочный файл модуля *Matrix.cpp*;
- *TimeCompare.hpp* — заголовочный файл модуля *TimeCompare.cpp*;
- *algoc.hpp* — заголовочный файл модуля *algos.cpp*;
- *GetCPUTime.hpp* — заголовочный файл модуля *GetCPUTime.cpp*;
- *Gen.hpp* — заголовочный файл модуля *Gen.cpp*.

2.3 Реализация алгоритмов

В листингах , представлены реализации алгоритмов поиска расстояний — .

Листинг 2.1 — Матричный алгоритм поиска пути Левенштейна

```
int non_rec_lev(const char* s1, const char* s2)
{
  int n = strlen(s1) + 1;
  int m = strlen(s2) + 1;
  if (n == 1)
  return m - 1;
  if (m == 1)
  return n - 1;
  data_t mtrx;
  mtrx.matrix = create_matrix(strlen(s1) + 1, strlen(s2) + 1);
  mtrx.n = strlen(s1) + 1;
  mtrx.m = strlen(s2) + 1;
  for (int i = 0; i < n; ++i)
  {
    for (int j = 0; j < m; ++j)
    {

      if (i == 0 && j == 0)
      mtrx.matrix[i][j] = 0;
      else if (i == 0)
      mtrx.matrix[i][j] = j;
      else if (j == 0)
```

```
23          mtrx.matrix[i][j] = i;
24          else
25          mtrx.matrix[i][j] = min_int(3, mtrx.matrix[i][j - 1] + 1,mtrx.
                matrix[i - 1][j] + 1,
26          mtrx.matrix[i - 1][j - 1] + (s1[i - 1] != s2[j - 1])
27          );
28        }
29    }
30    int result = mtrx.matrix[n - 1][m - 1];
31    free_matrix(&mtrx);
32    return result;
33 }
```

Листинг 2.2 — Матричный алгоритм поиска пути Дамерау-Левенштейна

```
1  int non_rec_dam_lev(const char* s1, const char* s2)
2  {
3    int n = strlen(s1) + 1;
4    int m = strlen(s2) + 1;
5    if (n == 1)
6    return m - 1;
7    if (m == 1)
8    return n - 1;
9    data_t mtrx;
10   mtrx.matrix = create_matrix(strlen(s1) + 1, strlen(s2) + 1);
11   mtrx.n = strlen(s1) + 1;
12   mtrx.m = strlen(s2) + 1;
13   for (int i = 0; i < n; ++i)
14   {
15     for (int j = 0; j < m; ++j)
16     {
17       if (i == 0 && j == 0)
18       mtrx.matrix[i][j] = 0;
19       else if (i == 0)
20       mtrx.matrix[i][j] = j;
21       else if (j == 0)
22       mtrx.matrix[i][j] = i;
23       else
24       {
```

```
25          mtrx.matrix[i][j] = min_int(3,
26          mtrx.matrix[i][j - 1] + 1,
27          mtrx.matrix[i - 1][j] + 1,
28          mtrx.matrix[i - 1][j - 1] + (s1[i - 1] != s2[j - 1])
29          );
30          if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2
                [j - 1])
31          mtrx.matrix[i][j] = min_int(2, mtrx.matrix[i][j], mtrx.matrix[i
                - 2][j - 2] + 1);
32        }
33      }
34    }
35    int result = mtrx.matrix[n - 1][m - 1];
36    free_matrix(&mtrx);
37    return result;
38  }
```

Листинг 2.3 — Рекурсивый алгоритм поиска пути Дамерау-Левенштейна

```
1  int rec_dam_lev(const char* s1, const char* s2)
2  {
3    return rec_dam_lev1(s1, s2, strlen(s1), strlen(s2));
4  }
5
6  int rec_dam_lev1(const char* s1, const char* s2, const int li1, const
       int li2)
7  {
8    int result = -1;
9    if (li1 == 0 || li2 == 0)
10   return std::abs(li1 - li2);
11   result = min_int(3,
12   rec_dam_lev1(s1, s2, li1 - 1, li2) + 1,rec_dam_lev1(s1, s2, li1, li2
       - 1) + 1,
13   rec_dam_lev1(s1, s2, li1 - 1, li2 - 1) + (s1[li1 - 1] != s2[li2 - 1])
       );
14   if (li1 > 1 && li2 > 1 && s1[li1 - 1] == s2[li2 - 2] && s1[li1 - 2]
       == s2[li2 - 1])
15   result = min_int(2,result,rec_dam_lev1(s1, s2, li1 - 2, li2 - 2) + 1)
       ;
```

```
16   return result;
17 }
```

Листинг 2.4 — Рекурсивый с кешем алгоритм поиска пути Дамерау-Левенштейна

```
1   int cache_dam_lev(const char* s1, const char* s2)
2   {
3     data_t cache;
4     cache.matrix = create_matrix(strlen(s1) + 1, strlen(s2) + 1);
5     cache.n = strlen(s1) + 1;
6     cache.m = strlen(s2) + 1;
7     fill_mtrx_with_inf(&cache);
8     int r = cache_dam_lev1(&cache, s1, s2, strlen(s1), strlen(s2));
9     free_matrix(&cache);
10    return r;
11  }
12
13  int cache_dam_lev1(data_t* cache, const char* s1, const char* s2,
      const int li1, const int li2)
14  {
15    if (cache->matrix[li1][li2] != LONG_MAX)
16    return cache->matrix[li1][li2];
17    if (li1 == 0 && li2 == 0)
18    {
19      cache->matrix[li1][li2] = 0;
20      return cache->matrix[li1][li2];
21    }
22    if (li1 == 0 && li2 > 0)
23    {
24      cache->matrix[li1][li2] = li2;
25      return li2;
26    }
27    if (li2 == 0 && li1 > 0)
28    {
29      cache->matrix[li1][li2] = li1;
30      return li1;
31    }
32    int r1 = 0, r2 = 0, r3 = 0;
33    r1 = cache_dam_lev1(cache, s1, s2, li1 - 1, li2) + 1;
```

```
34      r2 = cache_dam_lev1(cache, s1, s2, li1, li2 - 1) + 1;
35      r3 = cache_dam_lev1(cache, s1, s2, li1 - 1, li2 - 1) + (s1[li1 - 1]
            != s2[li2 - 1]);
36      cache->matrix[li1][li2] = min_int(3, r1, r2, r3);
37      int result = 0;
38      if (li1 > 1 && li2 > 1 && s1[li1 - 1] == s2[li2 - 2] && s1[li1 - 2]
            == s2[li2 - 1])
39      {
40        int r4 = 0;
41        r4 = cache_dam_lev1(cache, s1, s2, li1 - 2, li2 - 2) + 1;
42        cache->matrix[li1][li2] = min_int(2, cache->matrix[li1][li2], r4)
              ;
43      }
44      return cache->matrix[li1][li2];
45    }
```

2.4 Функциональное тестирование

В таблице 2.1 приведены тесты для функций, реализующих алгоритмы матричного умножения. Тесты для всех реализаций алгоритмов пройдены успешно.

Таблица 2.1 — Функциональные тесты

	Входные данные		Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1	”пустая строка”	”пустая строка”	0	0
2	”пустая строка”	слово	5	5
3	проверка	”пустая строка”	8	8
4	ремонт	емонт	1	1
5	гигиена	иена	3	3
6	нисан	автоваз	6	6
7	спасибо	пожалуйста	9	9
8	что	кто	1	1
9	ты	тыква	3	3
10	есть	кушать	4	4
11	abba	baab	3	2
12	abcba	bacab	4	2

Вывод

В данном разделе были представлены реализации следующих алгоритмов: стандартного матричного умножения, Винограда, Винограда с оптимизацией. Выполнено тестирование реализаций алгоритмов.

3. Исследовательская часть

В данном разделе будут приведены примеры работы программы, а также проведен сравнительный анализ процессорного времени и затрачиваемой памяти работы реализаций алгоритмов при различных ситуациях на основе полученных данных.

3.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени представлены далее:

- операционная система Windows 11 Pro Версия 22H2 (22621.674) [2];
- память 16 ГБ;
- процессор 11th Gen Intel(R) Core(TM) i5-11400 2.59 ГГц [3].

При тестировании компьютер был включен в сеть электропитания. Во время замеров процессорного времени устройство было нагружено только встроенными приложениями окружения, а также системой тестирования.

3.2 Демонстрация работы программы

На рисунке 3.1 представлен результат работы программы. На экран выводятся результаты заемров времени для разных размеров строк и разных видов алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна в мс.

Рис. 3.1 – Пример работы программы

3.3 Время выполнения реализаций алгоритмов

Как было сказано выше, используется функция замера процессорного времени GetProcessTimes(...) из библиотеки Windows.h.

Входные данные: строки размером от 10 до 500 элементов.

Результаты замеров времени работы реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна на различных входных данных (в мс) приведены в таблице 3.1.

Таблица 3.1 — Процессорное время работы реализаций алгоритмов

Размер	р. Л.(матр.)	р. Д-Л.(матр.)	р. Д-Л.(рек.)	р. Д-Л.(рек. с кешем)
10	0	0.015625	0	

Также на рисунке 3.2 приведены графические результаты замеров времени работы алгоритмов в зависимости от линейного размера входных строк.

Рис. 3.2 – Процессорное время вычислений

3.4 Затрачиваемая память при выполнении реализаций алгоритмов

3.5 Вывод

Заключение

Цель, которая была поставлена в начале лабораторной работы, была достигнута: изучены, реализованы и исследованы алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна.

В ходе выполнения лабораторной работы были решены все задачи:

1) изучены алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна;
2) разработаны алгоритмы поиска этих расстояний;
3) реализован каждый из данных алгоритмов;
4) проведены замеры процессорного времени для каждой из реализаций алгоритмов;
5) расчитана затрачиваемая реализованными алгоритмами пиковая память;
6) выполнен анализ полученных результатов;
7) по итогам работы составлен отчет.

Список использованных источников

1. GetProcessTimes function [Эл. ресурс]. Режим доступа: https://clck.ru/32NCYi (дата обращения: 13.10.2022).
2. Windows 11, version 22H2 [Эл. ресурс]. Режим доступа: https://clck.ru/32NCXx (дата обращения: 14.10.2022).
3. Процессор Intel® Core™ i7 [Эл. ресурс]. Режим доступа: https://clck.ru/yeQa8 (дата обращения: 14.10.2022).