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 ИУ «Информатика, искусственный интеллект и системы управления»

КАФЕДРА ИУ7 «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

по курсу «Анализ алгоритмов»

на тему:

«Редакционные расстояния»

Студент Рунов К. А.

Группа ИУ7-54Б

Преподаватели Волкова Л. Л., Строганов Д. В.

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Редакционные расстояния — расстояние Левенштейна и его модификация — расстояние Дамерау — Левенштейна — метрики сходства между двумя символьными последовательностями. Расстоянием в этих метриках считается минимальное число односимвольных преобразований (удаления, вставки, замены или транспозиции), необходимых для преобразования одной последовательности символов в другую.

Редакционные расстояния применяются

— для исправления ошибок в слове поискового запроса;

— в формах заполнения информации на сайтах;

— для распознавания рукописных символов;

— в базах данных. [1]

Целью данной лабораторной работы является изучение, реализация и исследование алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна.

Для достижения поставленной цели нужно решить следующие задачи:

1) описать алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна;

2) обосновать выбор средств реализации алгоритмов;

3) реализовать алгоритмы:

   — итеративный алгоритм нахождения расстояния Левенштейна,

   — итеративный алгоритм нахождения расстояния Дамерау — Левенштейна,

   — рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна,

   — рекурсивный с кешированием алгоритм нахождения расстояния Дамерау — Левенштейна;

4) провести сравнительный анализ алгоритмов по критериям:

- используемое процессорное время,
- максимальная затрачиваемая память;

5) описать и проанализировать полученные результаты в отчёте.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна — метрика, определяющая понятие расстояния между двумя последовательностями символов, как минимального количества редакторских операций вставки (I, от англ. insert), замены (R, от англ. replace) и удаления (D, от англ. delete), необходимых для преобразования одной строки в другую [2]. Для каждой операции должна быть определена её стоимость. Введём обозначения для стоимостей. Пусть:

1) $w(a, b)$ — цена замены символа a на b;

2) $w(\lambda, b)$ — цена вставки символа b;

3) $w(a, \lambda)$ — цена удаления символа a.

Определим стоимости операций:

$$w(a,b) = \begin{cases} 1, \text{ если } a \neq b; \\ 0, \text{ иначе.} \end{cases} \tag{1.1}$$

Отсутствие операций в случае совпадения символов будем обозначать за M (от англ. match).

Введём в рассмотрение функцию $D(S_1[1..i], S_2[1..j])$, значнием которой является редакционное расстояние между подстроками $S_1[1..i]$ и $S_2[1..j]$, где $S_1[1..i]$ — подстрока S_1 длины i. Так, если S_1 = "скат", то $S_1[1..0] = \lambda$, $S_1[1..1]$ = "с", $S_1[1..2]$ = "ск". Расстояние Левенштейна между строками S_1 и S_2 длин L_1 и L_2 соответственно вычисляется по рекуррентной формуле:

$$D(S_1[1..i], S_2[1..j]) = \\ = \begin{cases} \max(i,\ j), & i \cdot j = 0; \\ \min \begin{cases} D(S_1[1..i], S_2[1..j-1]) + 1, \\ D(S_1[1..i-1], S_2[1..j]) + 1, \\ D(S_1[1..i-1], S_2[1..j-1]) + w(S_1[i], S_2[j]), \end{cases} & i \cdot j \neq 0, \end{cases} \tag{1.2}$$

где $i = L_1$, $j = L_2$.

1.1.1 Итерационный алгоритм нахождения расстояния Левенштейна

Рекурсивная реализация алгоритма поиска расстояния Левенштейна малоэффективна по времени при больших L_1 и L_2, так как производится много повторных, лишних вычислений. Реализацию можно оптимизировать с помощью динамического программирования. Например, ввести матрицу размерности $(L_1 + 1) \times (L_2 + 1)$ и заполнять её промежуточными значениями $D(S_1[1..i], S_2[1..j])$, используя их затем по ходу вычислений. Значения в ячейках $[i][j]$ (i-я строка, j-й столбец) матрицы равны значениям $D(S_1[1..i], S_2[1..j])$ соответственно. Можно заметить, что всю матрицу для вычислений хранить не обязательно — двух строк будет достаточно.

1.2 Расстояние Дамерау — Левенштейна

Расстояние Дамерау — Левенштейна — метрика, которая определяет расстояние между двумя последовательностями символов, как и расстояние Левенштейна, но к исходному набору редакторских операций добавляется ещё одна — транспозиция (T, от англ. transposition). Операция транспозиции меняет местами соседние буквы в строке. Обозначим её стоимость: $w(ab, ba) = 1$.

Расстояние Дамерау — Левенштейна $\mathcal{D}(S_1, S_2)$ между строками S_1 и S_2 длин L_1 и L_2 соответственно может быть вычислено по рекуррентной фор-

муле:

$$
\begin{aligned}
&\mathcal{D}(S_1[1..i], S_2[1..j]) = \\
&= \begin{cases}
\max(i,\ j), & i \cdot j = 0; \\
\min \begin{cases}
\mathcal{D}(S_1[1..i], S_2[1..j-1]) + 1, \\
\mathcal{D}(S_1[1..i-1], S_2[1..j]) + 1, \\
\mathcal{D}(S_1[1..i-1], S_2[1..j-1]) + w(S_1[i], S_2[j]), \\
\begin{cases}
\mathcal{D}(S_1[1..i-2], S_2[1..j-2]) + 1, & \text{если } i > 1,\ j > 1, \\
& S_1[i] = S_2[j-1], \\
& S_1[i-1] = S_2[j]; \\
\infty, & \text{иначе,}
\end{cases}
\end{cases} & i \cdot j \neq 0,
\end{cases}
\end{aligned}
\tag{1.3}
$$

где $i = L_1$, $j = L_2$.

1.2.1 Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна реализует рекуррентную формулу (1.3). Таким образом, верно следующее:

1) $\mathcal{D}(\lambda, \lambda) = 0$, — для преобразования пустой строки в пустую строку требуется 0 операций вставки, замены, удаления и транспозиции;

2) $\mathcal{D}(S_1, \lambda) = |S_1|$ (длина S_1), — для преобразования строки S_1 в пустую строку требуется $|S_1|$ операций (удаления);

3) $\mathcal{D}(\lambda, S_2) = |S_2|$, — для преобразования пустой строки в строку S_2 требуется $|S_2|$ операций (вставки);

4) $\mathcal{D}(c_1, c_2) = \begin{cases} 1, & c_1 \neq c_2; \\ 0, & c_1 = c_2, \end{cases}$ — для преобразования одного символа в другой требуется 1 операция (замены), если символы отличаются, и 0 операций, если символы совпадают;

5) Для преобразования одной пары символов в другую:

$$\mathcal{D}(c_1c_2, c_3c_4) = \begin{cases} 0,\ c_1 = c_3,\ c_2 = c_4;\ //\ MM \\ 1,\ c_1 = c_3,\ c_2 \neq c_4;\ //\ MR \\ 1,\ c_1 \neq c_3,\ c_2 = c_4;\ //\ RM \\ 1,\ c_1 = c_4,\ c_2 = c_3;\ //\ T \\ 2,\ \text{иначе};\ //\ RR \end{cases}$$

6) Пусть $S_1 = S_1'c_2 = S_1''c_1c_2$, $S_2 = S_2'c_4 = S_2''c_3c_4$, где S_1' и S_2' — строки S_1 и S_2 без последних символов, S_1'' и S_2'' — строки S_1 и S_2 без двух последних символов, а c_1c_2 и c_3c_4 — пары их последних символов соответственно.

$$\mathcal{D}(S_1, S_2) = \min \begin{cases} \mathcal{D}(S_1'c_2, S_2') + 1, \\ \mathcal{D}(S_1', S_2'c_4) + 1, \\ \mathcal{D}(S_1', S_2') + w(c_2, c_4), \\ \mathcal{D}(S_1'', S_2'') + 1, & \text{если } c_2 = c_3,\ c_1 = c_4. \end{cases}$$

Можно заметить, что $\mathcal{D}(S_1, S_2)$ вычисляется как минимальная длина последовательности редакторских операций, которыми можно преобразовать строку S_1 в S_2' плюс цена вставки последнего символа из S_2, строку S_1' в S_2 плюс цена удаления последнего символа из S_1, строку S_1' в S_2' плюс цена замены последних символов строк S_1 и S_2, строку S_1'' в S_2'' плюс цена транспозиции двух последних символов строк, если она возможна. Для любых подстрок $\mathcal{S}_1$ и $\mathcal{S}_2$ строк S_1 и S_2, $\mathcal{D}(\mathcal{S}_1, \mathcal{S}_2)$ вычисляет минимальное количество редакторских операций. Следовательно, $\mathcal{D}(S_1, S_2)$ действительно считает расстояние Дамерау — Левенштейна для произвольных строк строк S_1 и S_2.

1.2.2 Рекурсивный с кешированием алгоритм нахождения расстояния Дамерау — Левенштейна

Рекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна прост, но его реализация на ЭВМ без дополнительных оптимизаций нерациональна, так как по нескольку раз считаются значения, которые уже были вычислены, а вычисление их может оказаться достаточно трудоёмким.

Идея возможной оптимизации состоит в следующем: хранить получаемые

по ходу выполнения алгоритма значения в матрице, а перед вычислением очередного — проверять, было ли оно посчитано ранее (заполнена ли соответствующая ячейка матрицы), и, если да, — брать его оттуда, не прибегая к повторным вычислениям.

1.2.3 Итерационный алгоритм нахождения расстояния Дамерау — Левенштейна

Как алгоритм поиска расстояния Левенштейна, так и алгоритм поиска расстояния Дамерау — Левенштейна можно реализовать нерекурсивно с помощью динамического программирования, используя матрицу расстояний.

Процесс вычисления значения ячейки матрицы показан на рисунке 1.

	s_{21}	s_{22}	s_{23}
s_{11}	N_T	$\cdots$	$\cdots$
s_{12}	$\vdots$	N_R	N_D
s_{13}	$\vdots$	N_I	$\boldsymbol{N}$

$$\boldsymbol{N} = \min \begin{cases} N_I + 1, \\ N_D + 1, \\ N_R + w(s_{13}, s_{23}), \\ \begin{cases} N_T + 1, & \text{если } s_{13} = s_{22},\ s_{12} = s_{23} \text{ и } s_{11},\ s_{21} \text{ существуют}, \\ \infty, & \text{иначе.} \end{cases} \end{cases}$$

Рисунок 1 – Вычисление расстояния Дамерау — Левенштейна с использованием матрицы

Таким образом, для нахождения расстояния Дамерау — Левенштейна хранить всю матрицу не обязательно — трёх строк будет достаточно. Заполнив последнее значение очередной строки, можно выполнить «циклическую прокрутку» строк матрицы вверх, после чего изменить значение первой ячейки последней теперь строки матрицы на значение первой ячейки предпоследней строки матрицы плюс 1, а затем продолжить заполнение матрицы по алгоритму, представленному на рисунке выше, перезаписывая значения ячеек.

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояний Левенштейна и Дамерау — Левенштейна. Поскольку данные расстояния могут быть вычислены с помощью рекуррентных формул, то алгоритмы могут быть реализованы как рекурсивно, так и итерационно.

2 Конструкторская часть

3 Технологическая часть

4 Исследовательская часть

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. А. Погорелов Д., М. Таразанов А. Сравнительный анализ алгоритмов редакционного расстояния Левенштейна и Дамерау-Левенштейна // Синергия Наук. 2019. URL: — Режим доступа: https://elibrary.ru/item.asp?id=36907767 (дата обращения 27.10.2023).

2. Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: «Наука», Доклады АН СССР, 1965. Т. 163. С. 845–848.

ПРИЛОЖЕНИЕ А

Листинг 1 – Создание матрицы для последующего использования в реализациях алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна

```
size_t **create_matrix(size_t n_rows, size_t n_columns)
{
    if (n_rows < 1 || n_columns < 1)
        return NULL;

    size_t *mem = (size_t*)malloc(n_rows * n_columns *
       sizeof(size_t));

    if (mem == NULL)
        return NULL;

    size_t **matrix = (size_t**)malloc(n_rows * sizeof(size_t*));

    if (matrix == NULL)
    {
        free(mem);
        return NULL;
    }

    for (size_t i = 0; i < n_rows; i++)
    {
        matrix[i] = mem + i * n_columns;
        matrix[i][0] = i;
    }

    for (size_t j = 1; j < n_columns; j++)
        matrix[0][j] = j;

    return matrix;
}
```

Листинг 2 – Освобождение памяти из-под созданной матрицы

```
void free_matrix(size_t **matrix, size_t *first_row)
{
    free(first_row);
    free(matrix);
```

```
5 }
```

Листинг 3 – Нахождение минимального числа из трёх и возврат указателя на него

```
1  size_t *min3(size_t *a, size_t *b, size_t *c)
2  {
3      size_t *result;
4      size_t min = std::min(*a, std::min(*b, *c));
5
6      if (min == *a)
7          result = a;
8      else if (min == *b)
9          result = b;
10     else
11         result = c;
12
13     return result;
14 }
```

Листинг 4 – Нахождение минимального числа из четырёх и возврат указателя на него

```
1  size_t *min4(size_t *a, size_t *b, size_t *c, size_t *d)
2  {
3      size_t *result;
4      size_t min = std::min(std::min(*a, *b), std::min(*c, *d));
5
6      if (min == *a)
7          result = a;
8      else if (min == *b)
9          result = b;
10     else if (min == *c)
11         result = c;
12     else
13         result = d;
14
15     return result;
16 }
```

Листинг 5 – Часть реализации итерационного алгоритма нахождения расстояния Левенштейна, участвующая в замерах времени выполнения реализаций исследуемых алгоритмов

```
size_t lev_ifm_helper(size_t **matrix, const wchar_t *s1, size_t
    len1, const wchar_t *s2, size_t len2)
{
    size_t result = 0;
    bool replace_skip_cond;
    size_t insert_cost, delete_cost, replace_cost, *who;

    for (size_t i = 1; i < len1; i++)
    {
        for (size_t j = 1; j < len2; j++)
        {
            insert_cost = matrix[i - 1][j] + 1;
            delete_cost = matrix[i][j - 1] + 1;
            replace_skip_cond = (s1[i] == s2[j]);
            replace_cost = matrix[i - 1][j - 1] +
                (replace_skip_cond ? 0 : 1);
            who = min3(&insert_cost, &delete_cost, &replace_cost);
            matrix[i][j] = *who;
        }
    }

    result = matrix[len1 - 1][len2 - 1];

    return result;
}
```

Листинг 6 – Реализация итерационного алгоритма нахождения расстояния Левенштейна

```
size_t levenshtein_iterative_full_matrix(const wchar_t *str1,
    size_t len1, const wchar_t *str2, size_t len2)
{
    if (len1 == 0) return len2;
    if (len2 == 0) return len1;

    ++len1;
    ++len2;
    const wchar_t *s1 = str1 - 1;
    const wchar_t *s2 = str2 - 1;
```

```
10
11      size_t **matrix = create_matrix(len1, len2);
12
13      if (matrix == NULL) return -1;
14
15      size_t result = lev_ifm_helper(matrix, s1, len1, s2, len2);
16
17      free_matrix(matrix, matrix[0]);
18
19      return result;
20  }
```

Листинг 7 – Часть реализации итерационного алгоритма нахождения расстояния Дамерау — Левенштейна, участвующая в замерах времени выполнения реализаций исследуемых алгоритмов

```
1  size_t damlev_ifm_helper(size_t **matrix, const wchar_t *s1,
       size_t len1, const wchar_t *s2, size_t len2)
2  {
3      size_t result = 0;
4      bool replace_skip_cond, swap_cond;
5      size_t insert_cost, delete_cost, replace_cost, swap_cost,
           *who;
6
7      for (size_t i = 1; i < len1; i++)
8      {
9          for (size_t j = 1; j < len2; j++)
10         {
11             insert_cost = matrix[i - 1][j] + 1;
12             delete_cost = matrix[i][j - 1] + 1;
13             replace_skip_cond = (s1[i] == s2[j]);
14             replace_cost = matrix[i - 1][j - 1] +
                   (replace_skip_cond ? 0 : 1);
15             if (i >= 2 && j >= 2) [[likely]]
16             {
17                 swap_cond = (s1[i] == s2[j - 1] && s1[i - 1] ==
                       s2[j]);
18                 swap_cost = swap_cond ? matrix[i - 2][j - 2] + 1
                       : U_INF; // U_INF = -1
19                 who = min4(&insert_cost, &delete_cost,
                       &replace_cost, &swap_cost);
20             }
```

```
21                 else
22                 {
23                     who = min3(&insert_cost, &delete_cost,
                           &replace_cost);
24                 }
25                 matrix[i][j] = *who;
26             }
27         }
28
29         result = matrix[len1 - 1][len2 - 1];
30
31         return result;
32 }
```

Листинг 8 – Реализация итерационного алгоритма нахождения расстояния Дамерау — Левенштейна

```
1 size_t damerau_levenshtein_iterative_full_matrix(const wchar_t
    *str1, size_t len1, const wchar_t *str2, size_t len2)
2 {
3     if (len1 == 0) return len2;
4     if (len2 == 0) return len1;
5
6     ++len1;
7     ++len2;
8     const wchar_t *s1 = str1 - 1;
9     const wchar_t *s2 = str2 - 1;
10
11    size_t **matrix = create_matrix(len1, len2);
12    if (matrix == NULL) return -1;
13
14    size_t result = damlev_ifm_helper(matrix, s1, len1, s2, len2);
15
16    free_matrix(matrix, matrix[0]);
17
18    return result;
19 }
```

Листинг 9 – Часть реализации рекурсивного с кешированием алгоритма нахождения расстояния Дамерау — Левенштейна, участвующая в замерах времени выполнения реализаций исследуемых алгоритмов

```
size_t damlev_rwc_helper(size_t **matrix, const wchar_t *str1,
    size_t len1, const wchar_t *str2, size_t len2)
{
    if (len1 == 0) return len2;
    if (len2 == 0) return len1;

    size_t i = len1 - 1;
    size_t j = len2 - 1;

    size_t insert = (((j > 0) && (matrix[i][j - 1] != U_INF))
        ? matrix[i][j - 1]
        : damlev_rwc_helper(matrix, str1, len1, str2, len2 - 1))
        + 1;

    size_t del = (((i > 0) && (matrix[i - 1][j] != U_INF))
        ? matrix[i - 1][j]
        : damlev_rwc_helper(matrix, str1, len1 - 1, str2, len2))
        + 1;

    size_t replace = (((i > 0 && j > 0) && (matrix[i - 1][j - 1]
       != U_INF))
        ? matrix[i - 1][j - 1]
        : damlev_rwc_helper(matrix, str1, len1 - 1, str2, len2 -
            1))
        + (str1[i] == str2[j] ? 0 : 1);

    size_t swap = U_INF;
    if (i > 1 && j > 1 && matrix[i - 2][j - 2] != U_INF &&
       (str1[i] == str2[j - 1] && str1[i - 1] == str2[j]))
    {
        swap = matrix[i - 2][j - 2] + 1;
    }
    else if (i >= 1 && j >= 1 && (str1[i] == str2[j - 1] &&
       str1[i - 1] == str2[j]))
    {
        swap = damlev_rwc_helper(matrix, str1, len1 - 2, str2,
            len2 - 2) + 1;
    }

    size_t result = *min4(&insert, &del, &replace, &swap);

```

```
36      if (matrix[i][j] == U_INF) matrix[i][j] = result;
37
38      return result;
39  }
```

Листинг 10 – Реализация рекурсивного с кешированием алгоритма нахождения расстояния Дамерау — Левенштейна

```
1  size_t damerau_levenshtein_recursive_with_cache(const wchar_t
       *str1, size_t len1, const wchar_t *str2, size_t len2)
2  {
3      if (len1 == 0) return len2;
4      if (len2 == 0) return len1;
5
6      size_t **matrix = create_matrix(len1, len2);
7
8      if (matrix == NULL) return -1;
9
10     for (size_t i = 0; i < len1; i++)
11         for (size_t j = 0; j < len2; j++)
12             matrix[i][j] = U_INF;
13
14     size_t result = damlev_rwc_helper(matrix, str1, len1, str2,
           len2);
15
16     free_matrix(matrix, matrix[0]);
17
18     return result;
19 }
```

Листинг 11 – Реализация рекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

```
1  size_t damerau_levenshtein_recursive_no_cache(const wchar_t
       *str1, size_t len1, const wchar_t *str2, size_t len2)
2  {
3      if (len1 == 0) return len2;
4      if (len2 == 0) return len1;
5
6      size_t insert = damerau_levenshtein_recursive_no_cache(str1,
           len1, str2, len2 - 1) + 1;
7      size_t del = damerau_levenshtein_recursive_no_cache(str1,
           len1 - 1, str2, len2) + 1;
```

```
8       size_t replace = damerau_levenshtein_recursive_no_cache(str1,
            len1 - 1, str2, len2 - 1)
9           + (str1[len1 - 1] == str2[len2 - 1] ? 0 : 1);
10      size_t swap = (len1 >= 2 && len2 >= 2)
11          ? (
12                  (str1[len1 - 1] == str2[len2 - 2] && str1[len1 -
                        2] == str2[len2 - 1])
13                  ? damerau_levenshtein_recursive_no_cache(str1,
                        len1 - 2, str2, len2 - 2) + 1
14                  : U_INF
15              )
16          : U_INF;
17
18      return *min4(&insert, &del, &replace, &swap);
19  }
```

ПРИЛОЖЕНИЕ Б