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»

Отчет по лабораторной работе №1
по дисциплине "Анализ алгоритмов"

Тема: Расстояния Левенштейна и Дамерау-Левенштейна

Студент: Сироткина П.Ю.

Группа: ИУ7-56Б

Оценка: _____

Преподаватель: Волкова Л.Л.

Москва — 2021 г.

Оглавление

Введение

Расстояние Левенштейна - минимальное количество операций вставки/удаления одного символа, а также замены одного символа на другой, необходимых для превращения одной строки в другую[1].

Расстояние Дамерау-Левенштейна вычисляется аналогично, с учетом добавления часто применяющейся операции операции, которую заметил Дамерау: транспозиция 2 соседних символов.

Применение расстояния Левенштйена:

- В сфере теоретической информатики: исправление ошибок и опечаток (например, в поисковых запросах);
- В сфере биоинформатики: сравнение белковых структур, генов, хромосом и тд, анализ иммунитета (цепочку молекул можно закодировать вполне определенным символом, т.к. на данный момент известных науке молекул меньше, чем количество букв в латинском алфавите).

Цель лабораторной работы:

1. Изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.
2. Оценка реализаций алгоритмов.

Задачи лабораторной работы:

1. Изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками.
2. Применение метода динамического программирования для матричной реализации указанных алгоритмов.
3. Получение практических навыков реализации указанных алгоритмов: матричные и рекурсивные версии.
4. Сравнительный анализ линейной и рекурсивной реализации выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти).
5. Экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма.
6. Описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как рассчетно-пояснительная записка к работе.

1 Аналитическая часть

Расстояние Левенштейна - минимальное количество операций вставки/удаления одного символа, а также замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Дамерау-Левенштейна вычисляется аналогично, с учетом добавления часто применяющейся операции операции, которую заметил Дамерау: транспозиция 2 соседних символов.

Задача по нахождению Расстояния Левенштейна и Дамерау-Левенштейна соответственно заключается в нахождении последовательности этих операций, стоимость которых будет минимальной.

Задаются базовые радакторские операции, т.н. правила, а также вводится понятие штрафа - цена одной операции.

Базовые операции:

1. **I (Insert)** - вставка символа. Штраф = 1.
2. **D (Delete)** - удаление символа. Штраф = 1.
3. **R (Replace)** - замена символа. Штраф = 1.
4. **M (Match)** - совпадение символа из 1 строки с символом из 2 строки. Штраф = 0.
5. **X (eXchange)** - транспозиция 2 соседних символов. Штраф = 1.

1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Пусть S1 и S2 - две строки над некоторым алфавитом(для определенности берется латинский алфавит).

Пусть λ обозначает пустую строку.

Вводится обозначение *length(S)* = длина строки S. S[1...i] обозначает подстроку строки S, включающую в себя первые i символов строки, а S[i] - i+1-ый символ строки S.

Тогда расстояние Левенштейна можно рассчитать по формуле (1.1):

$$D(S1[1..i], S2[1..j]) = \begin{cases} 0 & \text{i = 0, j = 0} \\ i & \text{i > 0, j = 0} \\ j & \text{i = 0, j > 0} \\ min\{ & \\ \quad D(S1[1..i], S2[1..j-1]) + 1 & \text{i > 0, j > 0} \\ \quad D(S1[1..i-1], S2[1..j]) + 1 & \\ \quad D(S1[1..i-1], S2[1..j-1]) + match(S1[i], S2[j])\} & \end{cases} \tag{1.1}$$

Функция match(a, b) = 0, если a = b, и 1 иначе.

Рекурсивный алгоритм вычисления расстояния Левенштейна является реализацией рекуррентной формулы 1.1. Расстоянием будет минимальное значение функции.

Рекуррентная формула составлена из следующий соображений:

- D(λ, λ) = 0 (первое выражение в системе). Для перевода пустой строки в пустую потребуется 0 операций;

- D(S, λ) = length(S) (второе выражение в системе). Для преобразования некоторой строки в пустую необходимо последовательно удалить все буквы в слове;
- D(λ, S) = length(S) (третье выражение в системе). Для преобразования пустой строки в некоторую непустую строку S понадобится length(S) раз воспользоваться операций вставки;
- Для преобразования некоторой непустой строки S1 (length(S1) = i) в непустую строку S2 (length(S2) = j) потребуется выполнить некую последовательность операций вставки, удаления и замены. Цена преобразования строки S1 в строку S2 может быть выражена как:
  - Стоимость преобразования S1[1..i] в S2[1..j-1] + Цена операции вставки (т.е. 1), которая необходима для преобразования S2[1..j-1] в S2[1..j];
  - Стоимость преобразования S1[1..i-1] в S2[1..j] + Цена операции удаления (т.е. 1), которая необходима для преобразования S1[1..i] в S1[1..i-1];
  - Стоимость преобразования S1[1..i-1] в S2[1..j-1] + Цена операции определения совпадения (0, если S1[i] = S2[j], 1 иначе), которая необходима для преобразования S1[1..i] в S1[1..i-1] и S2[1..j] в S2[1..j-1].

1.2 Матричный алгоритм нахождения расстояния Левенштейна

Прямая реализация формулы 1.1 может быть малоэффективна по времени исполнения при больших i и j, т.к. промежуточные значения D(S1[1..i], S2[1..j]) вычисляются заново много раз.

Например, если представить этот процесс в виде дерева рекурсивных вызовов, то многие поддеревья будут содержать в себе другие идентичные поддеревья и вычислять их заново смысла нет.

Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В этом случае алгоритм является реализацией построчного заполнения некоторой матрицы A размерами length(S1) x length(S2) знаечниями D(S1[1..i], S2[1..j]).

1.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с заполнением матрицы

Рекурсивный алгоритм заполнения можно оптимизировать по времени исполнения с использованием матричного алгоритма.

Суть данного метода заключается в параллельном заполнении матрицы при выполнении рекурсии. В случае, если при выполнении рекурсии обрабатываются подстроки, которые ранее не были обработаны, то алгоритм делает прогон данных для них и заносит результат в матрицу, а если подстроки уже обрабатывались, то аолгритм ничего с ними не делает и переходит к обработке следующих пар подстрок.

1.4 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна можно рассчитать по формуле (1.2):

$$D(S1[1..i], S2[1..j]) = \begin{cases} max(i,j) & \min(\text{i,j}) = 0 \\ min\{ & \\ \quad D(S1[1..i], S2[1..j-1]) + 1 & \text{i} > 0,\ \text{j} > 0 \\ \quad D(S1[1..i-1], S2[1..j]) + 1 & \\ \quad D(S1[1..i-1], S2[1..j-1]) + match(S1[i], S2[j]) & \\ \quad \begin{cases} D(S1[1..i-2], S2[1..j-2]) + 1, & \text{если } i,j > 1; \\ & S1[i] = S2[j-1]; \\ & S2[j] = S1[i-1] \\ \infty, & \text{иначе} \end{cases} & \\ \} & \end{cases} \tag{1.2}$$

Формула выводится исходя из тех же соображений, что и 1.1, дополнительно рассматривается случай перестановки двух последовательных символов (4 вариант в поиске минимума).

Бесконечность в случае "иначе"в качестве ответа означает то, что при выборе максимума этот случай рассматриваться не будет, т.к. первые 3 выражения заведомо не дадут результат в виде бесконечноти.

1.5 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, задача которых - определить минимальное количество операций вставки/удаления одного символа, а также замены одного символа на другой и транспозиции двух пар символов, необходимых для превращения одной строки в другую.

Формулы для вычисления задаются в рекурсивном виде (см. формулы 1.1 и 1.2). Как известно, рекурсия - часто не самый эффективный способ решения, на больших данных будет затрачиваться большое количество памяти и времени, поэтому был рассмотрен способ оптимизации вычислений - использование матрицы для хранения промежуточного ответа.

2 Конструкторская часть

В данном разделе представлены схемы алгоритмов, описанных в аналитическом разделе.

2.1 Схемы алгоритма Левенштейна

На рисунке 2.1 представлена схема рекурсивного алгоритма нахождения расстояния Левенштейна.

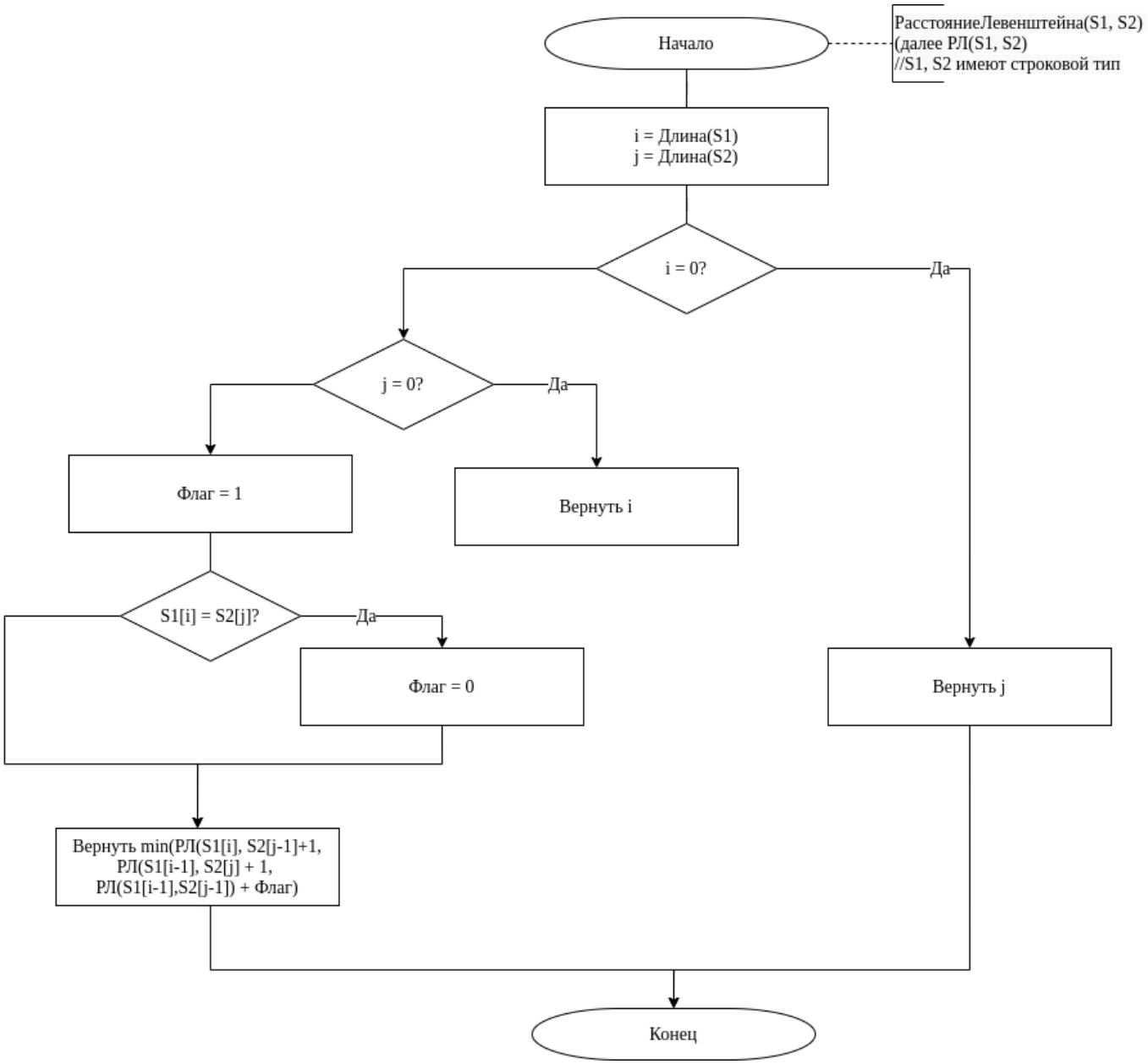

Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

На рисунке 2.2 представлена схема рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы.

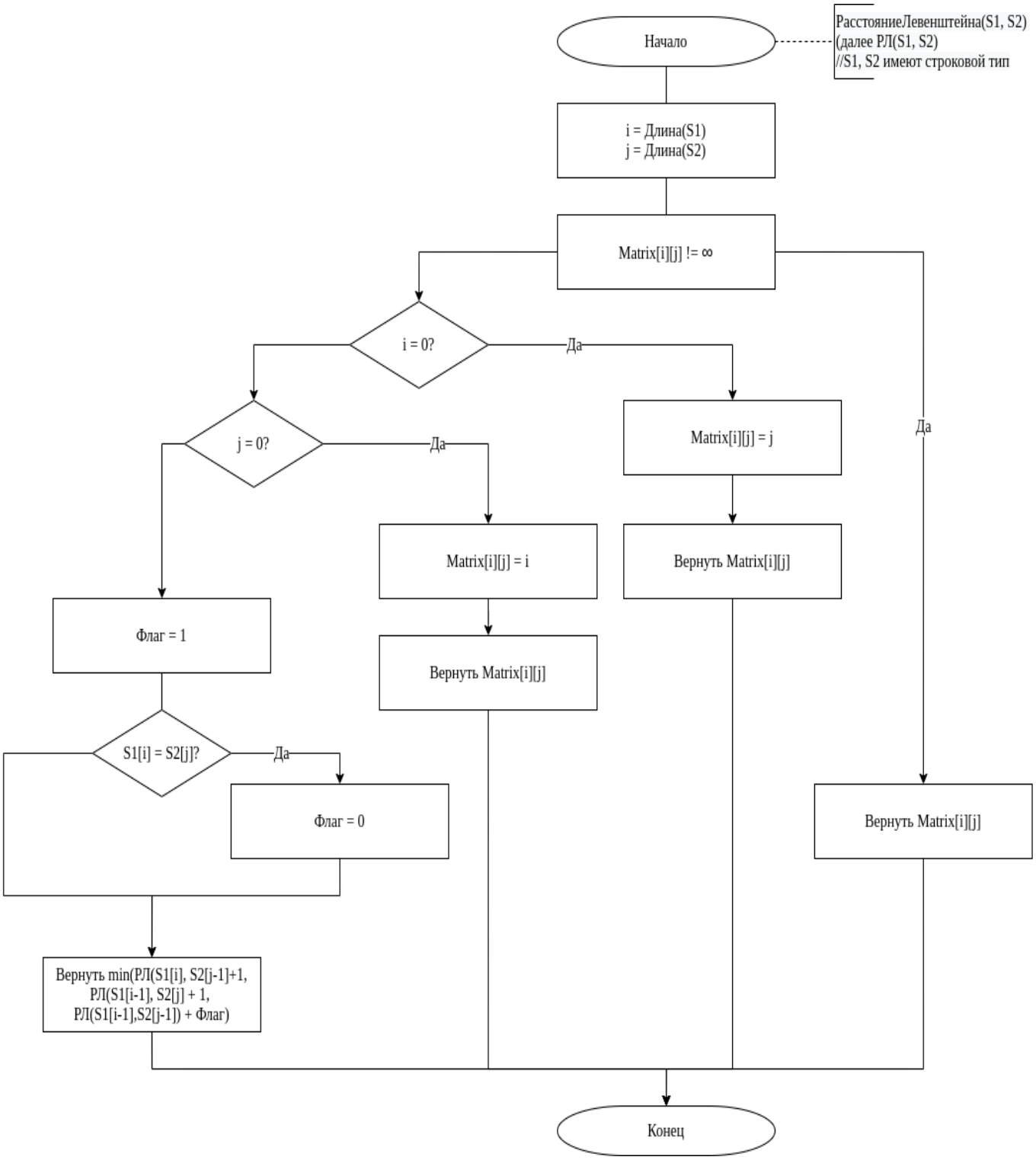

Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы

На рисунке 2.3 представлена схема матричного(итерационного) алгоритма нахождения расстояния Левенштейна.

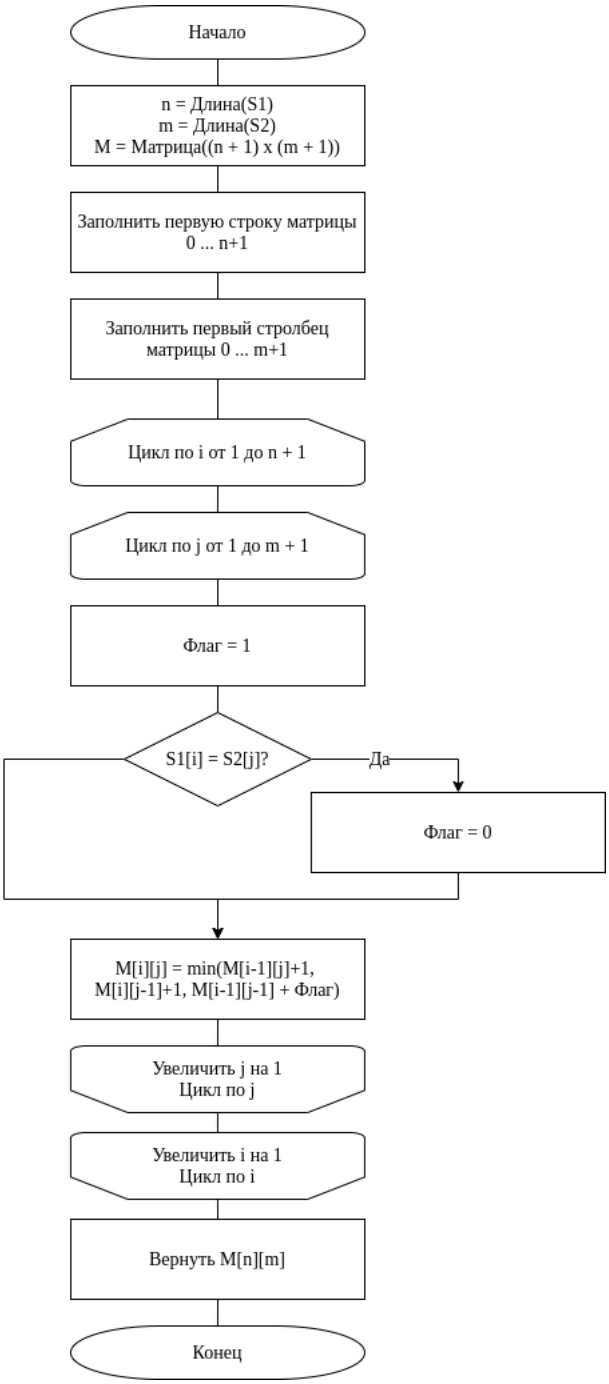

Рис. 2.3: Схема матричного(итерационного) алгоритма нахождения расстояния Левенштейна

2.2 Схема алгоритма Дамерау-Левенштейна

На рисунке 2.4 представлена схема алгоритма нахождения расстояния Дамерау-Левенштейна.

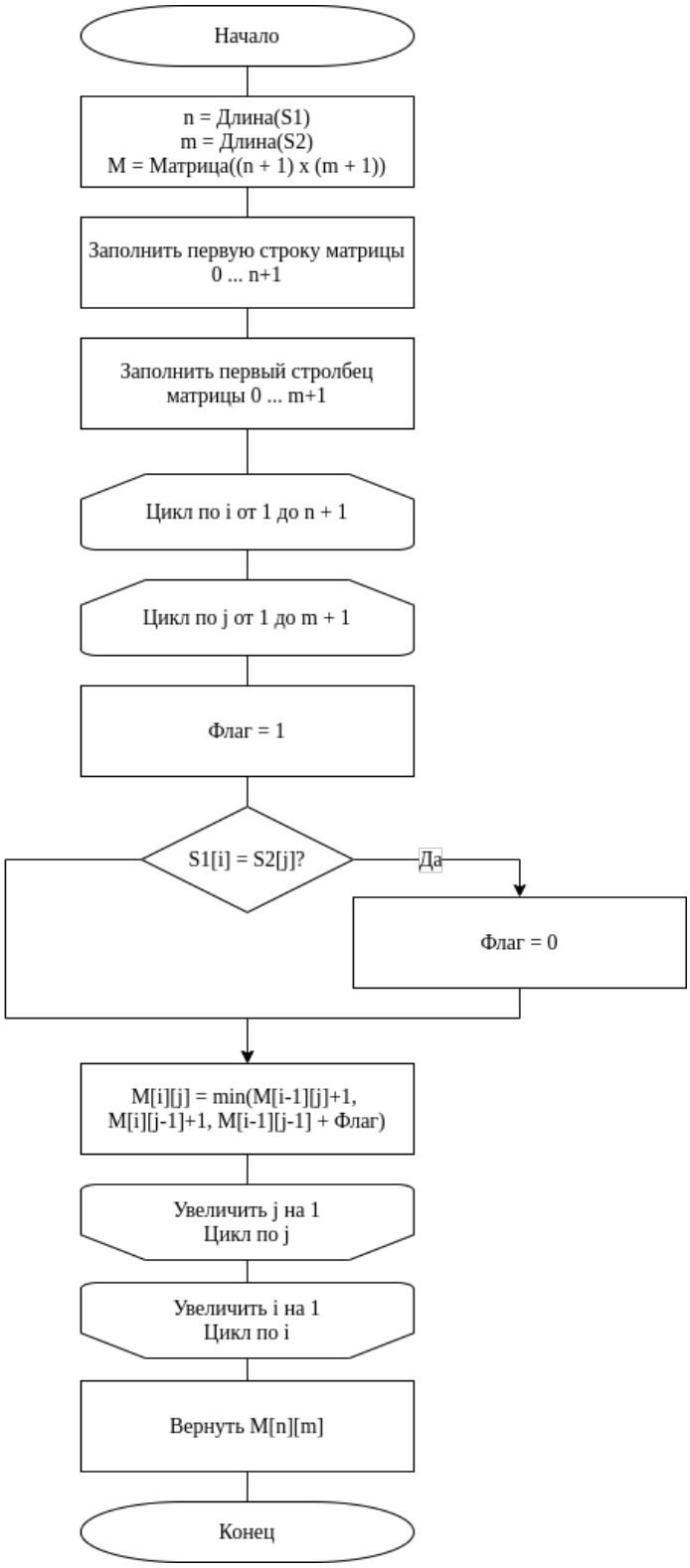

Рис. 2.4: Схема алгоритма нахождения расстояния Дамерау-Левенштейна

2.3 Вывод

В этом разделе представлены схемы алгоритмов Левенштейна и Дамерау-Левенштейна, основанные на теоретических данных, полученных из аналитического раздела.

3 Технологическая часть

В данном разделе приведны требования к программному обеспечению, средства реализации и листинги кода.

3.1 Требования к ПО

1. На вход подаются две строки в любой раскладке, в том числе и пустые;
2. Результат работы программы - искомое расстояние для всех методов, а также матрицы расстояний для всех методов, кроме рекурсивного;
3. Должна быть возможность просмотра потраченного времени и памяти.

3.2 Средства реализации

В качестве языка программирования для реализации лабораторной работы был выбран язык C. Выбор этого языка обусловен его быстродействием и эффективностью, это легкочитаемый, лаконичный и гибкий язык. Также выбор обусловлен моим личным желанием получить больше практики написания программ на этом языке.

3.3 Листинг кода

Листинг 3.1: Нахождение расстояния Левенштейна рекурсивно

```
int lev_rec(const char *s1, const char *s2)
{
    int i = strlen(s1), j = strlen(s2);

    if (i == 0 || j == 0)
        return max(i, j);

    int match = (s1[i - 1] == s2[j - 1]) ? 0 : 1;

    return min_from_three(lev_rec(s1, substr(s2, 0, j - 1)) + 1,
                          lev_rec(substr(s1, 0, i - 1), s2) + 1,
                          lev_rec(substr(s1, 0, i - 1), substr(s2, 0, j - 1)) + match);
}
```

Листинг 3.2: Нахождение расстояния Левенштейна рекурсивно с использованием кеширования

```
int lev_rec_matrix(const char *s1, const char *s2)
{
    int i = strlen(s1), j = strlen(s2);

    int **matrix = allocate_matrix(i + 1, j + 1);
    fill_matrix_int_max(matrix, i + 1, j + 1);

    int dist = get_lev_dist(s1, s2, matrix);
```

```
    free_matrix(matrix, i + 1);

    return dist;
}

int get_lev_dist(const char *s1, const char *s2, int **matrix)
{
    int i = strlen(s1), j = strlen(s2);

    if (matrix[i][j] != INT_MAX)
        return matrix[i][j];

    if (i == 0)
    {
        matrix[i][j] = j;
        return matrix[i][j];
    }

    if (i > 0 && j == 0)
    {
        matrix[i][j] = i;
        return matrix[i][j];
    }

    int match = (s1[i - 1] == s2[j - 1]) ? 0 : 1;

    matrix[i][j] = min_from_three(
                get_lev_dist(s1, substr(s2, 0, j - 1), matrix) + 1,
                get_lev_dist(substr(s1, 0, i - 1), s2, matrix) + 1,
                get_lev_dist(substr(s1, 0, i - 1), substr(s2, 0, j - 1), matrix)
                + match);

    return matrix[i][j];
}
```

Листинг 3.3: Нахождение расстояния Левенштейна матрично(итерационно)

```
int lev_iter(const char *s1, const char *s2)
{
    int n = strlen(s1), m = strlen(s2);

    int **matrix = allocate_matrix(n + 1, m + 1);
    fill_matrix_int_max(matrix, n + 1, m + 1);

    for (int i = 0; i < n + 1; i++)
        matrix[i][0] = i;

    for (int j = 0; j < m + 1; j++)
        matrix[0][j] = j;

```

```
14      int insert_dist, delete_dist, match_dist, match;
15
16      for (int i = 1; i < n + 1; i++)
17          for (int j = 1; j < m + 1; j++)
18          {
19              match = (s1[i - 1] == s2[j - 1]) ? 0 : 1;
20
21              insert_dist = matrix[i][j - 1] + 1;
22              delete_dist = matrix[i - 1][j] + 1;
23              match_dist = matrix[i - 1][j - 1] + match;
24
25              matrix[i][j] = min_from_three(insert_dist, delete_dist, match_dist);
26          }
27
28      int dist = matrix[n][m];
29      free_matrix(matrix, n + 1);
30
31      return dist;
32  }
```

Листинг 3.4: Нахождение расстояния Дамерау-Левенштейна рекурсивно

```
1   int dam_lev_rec(const char *s1, const char *s2)
2   {
3       int i = strlen(s1), j = strlen(s2);
4
5       if (i == 0 || j == 0)
6           return max(i, j);
7
8       int match = (s1[i - 1] == s2[j - 1]) ? 0 : 1;
9
10      int res = min_from_three(
11                          dam_lev_rec(s1, substr(s2, 0, j - 1)) + 1,
12                          dam_lev_rec(substr(s1, 0, i - 1), s2) + 1,
13                          dam_lev_rec(substr(s1, 0, i - 1), substr(s2, 0, j - 1)) + match);
14
15      if (i > 1 && j > 1 && s1[i] == s2[j - 1] && s1[i - 1] == s2[j])
16      {
17          res = min(dam_lev_rec(substr(s1, 0, i - 2), substr(s2, 0, j - 2)) + 1, res);
18      }
19
20      return res;
21  }
```

Листинг 3.5: Нахождение расстояния Дамерау-Левенштейна рекурсивно с использованием кеширования

```
1   int dam_lev_rec_matrix(const char *s1, const char *s2)
2   {
3       int i = strlen(s1), j = strlen(s2);
4
```

```
5       int **matrix = allocate_matrix(i  + 1, j + 1);
6       fill_matrix_int_max(matrix, i + 1, j + 1);
7
8       int dist = get_lev_dist(s1, s2, matrix);
9       free_matrix(matrix, i + 1);
10
11      return dist;
12  }
13
14  int get_dam_lev_dist(const char *s1, const char *s2, int **matrix)
15  {
16      int i = strlen(s1), j = strlen(s2);
17
18      if (matrix[i][j] != INT_MAX)
19          return matrix[i][j];
20
21      if (i == 0)
22      {
23          matrix[i][j] = j;
24          return matrix[i][j];
25      }
26
27      if (i > 0 && j == 0)
28      {
29          matrix[i][j] = i;
30          return matrix[i][j];
31      }
32
33      int match = (s1[i - 1] == s2[j - 1]) ? 0 : 1;
34
35      int res = min_from_three(
36                  get_dam_lev_dist(s1, substr(s2, 0, j - 1), matrix) + 1,
37                  get_dam_lev_dist(substr(s1, 0, i - 1), s2, matrix) + 1,
38                  get_dam_lev_dist(substr(s1, 0, i - 1), substr(s2, 0, j - 1), matrix)
39                  + match);
40
41      if (i > 1 && j > 1 && s1[i] == s2[j - 1] && s1[i - 1] == s2[j])
42          res = min(get_dam_lev_dist(substr(s1, 0, i - 2),
43                                     substr(s2, 0, j - 2), matrix), res);
44
45      matrix[i][j] = res;
46
47      return matrix[i][j];
48  }
```

Листинг 3.6: Нахождение расстояния Дамерау-Левенштейна матрично(итерационно)

```
1   int dam_lev_iter(const char *s1, const char *s2)
2   {
3       int n = strlen(s1), m = strlen(s2);
```

```
4
5       int **matrix = allocate_matrix(n  + 1, m + 1);
6       fill_matrix_int_max(matrix, n + 1, m + 1);
7
8       for (int i = 0; i < n + 1; i++)
9           matrix[i][0] = i;
10
11      for (int j = 0; j < m + 1; j++)
12          matrix[0][j] = j;
13
14      int insert_dist, delete_dist, match_dist, exchange_dist;
15      int match;
16
17      for (int i = 1; i < n + 1; i++)
18          for (int j = 1; j < m + 1; j++)
19          {
20              match = (s1[i - 1] == s2[j - 1]) ? 0 : 1;
21
22              if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1])
23                  exchange_dist = matrix[i - 2][j - 2] + 1;
24              else
25                  exchange_dist = INT_MAX;
26
27              insert_dist = matrix[i][j - 1] + 1;
28              delete_dist = matrix[i - 1][j] + 1;
29              match_dist = matrix[i - 1][j - 1] + match;
30
31              matrix[i][j] = min_from_four(insert_dist, delete_dist,
32                                           match_dist, exchange_dist);
33          }
34
35      int dist = matrix[n][m];
36      free_matrix(matrix, n + 1);
37
38      return dist;
39  }
```

Листинг 3.7: Вспомогательные функции, упомянутые выше

```
1  int min(int a, int b)
2  {
3      if (a < b)
4          return a;
5      return b;
6  }
7
8  int max(int a, int b)
9  {
10     if (a > b)
11         return a;
```

```
12      return b;
13  }
14
15  int min_from_three(int a, int b, int c)
16  {
17      return min(a, min(b, c));
18  }
19
20  int min_from_four(int a, int b, int c, int d)
21  {
22      return min(a, min_from_three(c, b, d));
23  }
24
25  char* substr(const char *str, int start, int len)
26  {
27      char *s;
28
29      s = malloc((len - start + 1) * sizeof(char));
30
31      for (int i = start; i < len; i++)
32          s[i] = str[i + start];
33
34      return s;
35  }
36
37  int **allocate_matrix(int m, int n)
38  {
39      int **data = calloc(m, sizeof(int*));
40      if (!data)
41          return NULL;
42      for (int i = 0; i < m; i++)
43      {
44          data[i] = malloc(n * sizeof(int));
45          if (!data[i])
46          {
47              free_matrix(data, m);
48              return NULL;
49          }
50      }
51      return data;
52  }
53
54  void fill_matrix_int_max(int **matrix, int n, int m)
55  {
56      for (int i = 0; i < n; i++)
57          for (int j = 0; j < m; j++)
58              matrix[i][j] = INT_MAX;
59  }
60
```

```
61 void free_matrix(int **matrix, int size)
62 {
63     for (int i = 0; i < size; i++)
64         free(matrix[i]);
65     free(matrix);
66 }
```

3.4 Тестовые данные

В таблице 3.1 приведены тестовые данные, на которых было протестировано разработанное ПО. Все тесты были успешно пройдены.

Таблица 3.1: Таблица тестовых данных

№	Первое слово	Второе слово	Левенштейн	Дамерау-Левенштейн
1			0	0
2		love	4	4
3	love		4	4
4	cow	woc	2	2
5	let	letter	5	5
6	qwerty	qwe	3	3
7	cat	cta	2	1
8	head	ehda	3	2
9	death	health	2	2
10	monday	monday	0	0

3.5 Вывод

В данном разделе были разработаны исходные коды выбранных алгоритмов: вычисления расстояния Левенштейна и Дамерау-Левенштейна рекурсивно, рекурсивно с заполнением матрицы и матрично.

4 Исследовательская часть

В данном разделе представлен пользовательский интерфейс, а также проведена оценка эффективности алгоритмов.

4.1 Пример работы

На рисунке 4.1 приведен пример работы программы.

```
polina@polina-IdeaPad-5-14ARE05:~/aa$ ./app.exe
Выберите действие:

1) Ввести две строки и посчитать расстояние для них;
2) Показать сравнительный анализ эффективности алгоритмов;
3) Выход.

Ответ: 1

Первая строка: death

Вторая строка: health

Расстояние Левенштейна: рекурсивный алгоритм: 2

Расстояние Левенштейна: рекурсивный алгоритм с кешированием: 2

Расстояние Левенштейна: матричный(итеративный) алгоритм: 2

Расстояние Дамерау-Левенштейна: 2

Матрица по Левенштейну:

0 1 2 3 4 5 6
1 1 2 3 4 5 6
2 2 1 2 3 4 5
3 3 2 1 2 3 4
4 4 3 2 2 2 3
5 4 4 3 3 3 2

Матрица по Дамерау-Левенштейну:

0 1 2 3 4 5 6
1 1 2 3 4 5 6
2 2 1 2 3 4 5
3 3 2 1 2 3 4
4 4 3 2 2 2 3
5 4 4 3 3 3 2

Выберите действие:

1) Ввести две строки и посчитать расстояние для них;
2) Показать сравнительный анализ эффективности алгоритмов;
3) Выход.

Ответ:
```

Рис. 4.1: Пример работы программы

4.2 Технические характеристики

Технические характеритиски машины, на которой выполнялось тестирование:

- Операционная система: Ubuntu[3] Linux[4] 20.04 64-bit.
- Оперативная память: 16 Gb.
- Процессор: AMD(R) Ryzen(TM)[5] 5 4500U CPU @ 2.3 CHz

4.3 Время выполнения алгоритмов

Время выполнения агоритмов (процессорное) замерялось с помощью ассемблерной вставки, которая ведет посчет тиков процессора[2]:

Листинг 4.1: "Ассемблерная вставка для замера тиков процессора"

```
uint64_t tick(void)
{
    uint32_t high, low;
    __asm__ __volatile__(
        "rdtsc\n"
        "movl %%edx, %0\n"
        "movl %%eax, %1\n"
        : "=r"(high), "=r"(low)::"%rax", "%rbx", "%rcx", "%rdx");

    uint64_t ticks = ((uint64_t)high << 32) | low;

    return ticks;
}
```

В таблице 4.1 приведены замеры процессорного времени для каждого из алгоритмов (Л - Левенштйе, ДЛ - Дамерау-Левенштейн):

Таблица 4.1: Таблица замеров процессорного времени (в тиках).

Длина строки	Л: Рекурсия	Д: Рекурсия	Л: Кэширование	Д: Кэширование	Л: Матрица	Д: Матрица
5	2 096 910	9 649 916	21 440	21 302	2 180	2 982
10	-	-	79 930	101 161	6 445	9 430
20	-	-	405 981	455 066	29 503	34 119
40	-	-	2 347 010	2 672 620	102 847	149 847
80	-	-	15 898 429	17 552 178	380 299	561 260
160	-	-	114 139 866	95 198 994	1 365 697	1 237 296

На рисунке 4.2 изображена зависимость процессорного времени работы рекурсивного алгоритма с кэшированием и матричного алгоритма от длины строк для итеративного алгоритма и алгоритма с кэшированием (обе строки пологаются одинаковой длины):

На рисунке 4.3 изображена зависимость процессорного времени работы рекурсивного алгоритма с кэшированием для нахождения расстояния Левенштейна и Дамерау-Левенштейна от длины строк (обе строки пологаются одинаковой длины):

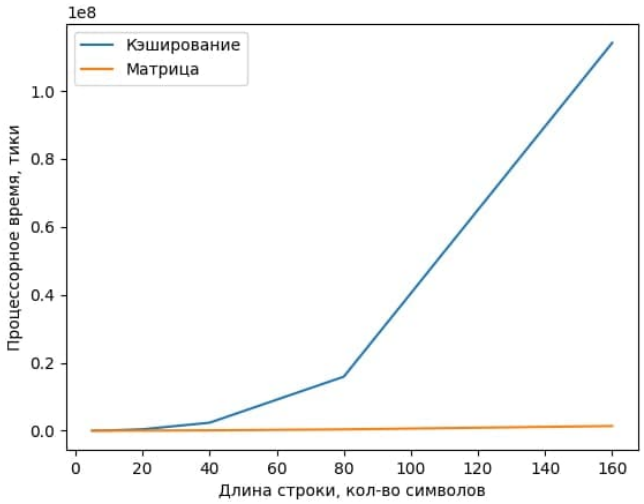

Рис. 4.2: Зависимость процессорного времени от длины
строк для итеративного алгоритма и алгоритма с кэшированием

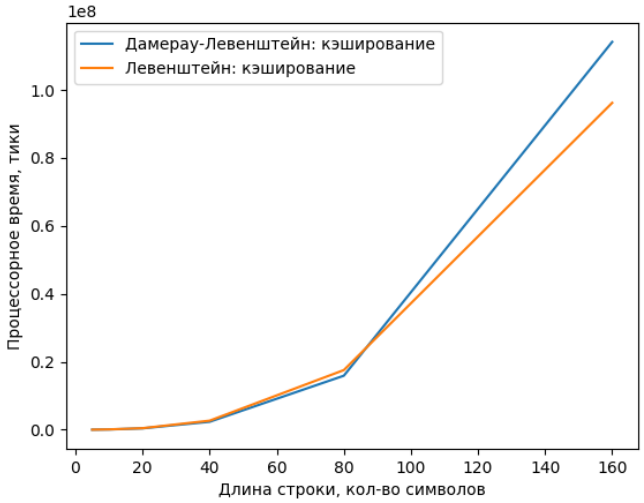

Рис. 4.3: Зависимость процессорного времени работы рекурсивного
алгоритма с кэшированием для нахождения расстояния
Левенштейна и Дамерау-Левенштейна от длины строк

4.4 Использование памяти

Алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются друг от друга с точки зрения использования памяти, поэтому достаточно рассмотреть разницу рекурсивной и матричной реализации одного из алгоритмов.

Максимальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящих строк. Поэтому максимальный объем памяти равен:

$$(Length(S1) + Length(S2)) \cdot (2 * SizeOf(*String) + 3 * SizeOf(Int)) \tag{4.1}$$

Length() - оператор вычисления длины строки, SizeOf() - оператор вычисления длины переменной в байтах,

*String - тип данных указатель на строку, Int - целочисленный тип данных.

Использование памяти при итеративной реализации теоретически равно:

$$(Length(S1)+1)\cdot(Length(S2)+1)\cdot SizeOf(Int)+2*SizeOf(*String)+3*SizeOf(Int) \tag{4.2}$$

4.5 Вывод

Рекурсивный вариант реализации нахождения расстояния Левенштейна работает существенно дольше итеративных реализаций: на строках длиной 5 рекурсивная реализация нахождения расстояния Левенштейна работает в 1000 раз медленнее итеративной реализации и в 100 раз медленее рекурсивной реализации с кэшированием. Стоит заметить, что такой большой проигрыш происходит при длине строки всего в 5 символов, следовательно, на более длинных строках нет смысла использовать рекурсивную реализацию. Рекурсивный вариант Дамерау-Левенштейна работает еще медленее: в 4-5 раз медленее аналогичной реализации Левенштейна.

В свою очередь, рекурсивная реализация проигрывает итеративной: на строках длиной 160 кэширование работает примерно в 100 раз медленее.

Однако, итеративные памяти проигрывают по памяти: в итеративной реализации максимальный размер памяти растет как произведение длин строк, в то время как в рекурсивной реализации - как сумма длин строк.

Заключение

В ходе проделанной работы были выполнены все поставленные задачи: был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна, а также проведена оценка реализации алгоритмов с описанием и обоснованием проделанных результатов.

Экспериментально были установлены различия в производительности различных алгоритмов нахождения редакционного расстояния. Рекурсивный вариант реализации нахождения расстояния Левенштейна работает существенно дольше итеративных реализаций: на строках длиной 5 рекурсивная реализация нахождения расстояния Левенштейна работает в 1000 раз медленнее итеративной реализации и в 100 раз медленее рекурсивной реализации с кэшированием. Стоит заметить, что такой большой проигрыш происходит при длине строки всего в 5 символов, следовательно, на более длинных строках нет смысла использовать рекурсивную реализацию. Рекурсивный вариант Дамерау-Левенштейна работает еще медленее: в 4-5 раз медленее аналогичной реализации Левенштейна.

Теоретически было рассчитано использование памяти различных алгоритмов нахождения редакционного расстояния. Матричные алгоритмы потребляют больше памяти, чем рекурсивные, за счет выделения памяти под матрицу и введения дополнительных промежуточных переменных. Если использовать статическую матрицу в реализации рекурсивного метода с кэшированием, то это приведет к множественному копированию этой матрицы при рекурсивных вызовах, что нельзя назвать преимуществом алгоритма.

Список использованных источников

1. Вычисление редакционного расстояния [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/117063/. Дата обращения: 16.09.2021.
2. C/C++: как измерять процессорное время [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/282301/. Дата обращения: 16.09.2021.
3. Ubuntu [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Ubuntu. Дата обращения: 18.09.2021.
4. Linux: [Электронный ресурс]. Режим доступа: https://ru.wikipedia.org/wiki/Linux. Дата обращения: 18.09.2021.
5. Процессор AMD Ryzen(TM) 5 [Электронный ресурс]. Режим доступа: https://www.notebookcheck-ru.com/Obzor-noutbuka-Lenovo-IdeaPad-5-14ARE05.491399.0.html. Дата обращения: 21.09.2021.