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 № 1
по дисциплине "Анализ алгоритмов"

Тема Алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна

Студент Калашников С.Д.

Группа ИУ7-53Б

Преподаватель Волкова Л.Л., Строганов Ю.В.

Москва, 2022

СОДЕРЖАНИЕ

Введение **4**

1 Аналитическая часть **6**

1.1 Расстояние Левенштейна 6

1.1.1 Матричный алгоритм нахождения расстояния 7

1.2 Расстояние Дамерау-Левенштейна 8

1.2.1 Рекурсивный алгоритм нахождения расстояния 9

1.2.2 Матричный алгоритм нахождения расстояния 10

1.2.3 Рекурсивный алгоритм нахождения расстояния с использованием кеша 11

2 Конструкторская часть **12**

2.1 Описание алгоритмов 12

2.2 Использование памяти 20

3 Технологическая часть **22**

3.1 Требования к программному обеспечению 22

3.2 Средства реализации 22

3.3 Сведения о модулях программы 22

3.4 Реализация алгоритмов 23

3.5 Функциональное тестирование 28

4 Исследовательская часть **29**

4.1 Технические характеристики 29

4.2 Демонстрация работы программы 29

4.3 Время выполнения реализаций алгоритмов 30

4.4 Вывод 36

Заключение **37**

***** **39**

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) расчитать затрачиваемую реализованными алгоритмами память;
4) реализовать каждый из данных алгоритмов;
5) провести замеры процессорного времени для каждой из реализаций алгоритмов;
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 — вставка символа в произвольной позиции ($w(\lambda, b) = 1$);
- D — удаление символа в произвольной позиции ($w(\lambda, b) = 1$);
- R — замена символа на другой ($w(a, b) = 1,\ a \neq b$);
- M — совпадение двух символов ($w(a, a) = 0$).

С учетом введенных обозначений, расстояние Левенштейна может быть подсчитано по формуле:

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

Функция $m(a, b)$ из 1.1 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:

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

Функция $m(S1[i], S2[j])$ из 1.4 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:

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

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:

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

Функция $m(S1[i], S2[j])$ из 1.8 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.

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 Конструкторская часть

2.1 Описание алгоритмов

В данном разделе будут рассмотрены схемы алгоритмов поиска расстояний — Левенштейна (рис. 2.1, 2.2), Дамерау-Левенштейна матричным способом (рис. 2.3, 2.4), Дамерау-Левенштейна рекурсивным способом (рис. 2.5), Дамерау-Левенштейна рекурсивным с кеширование способом (рис. 2.6, 2.7).

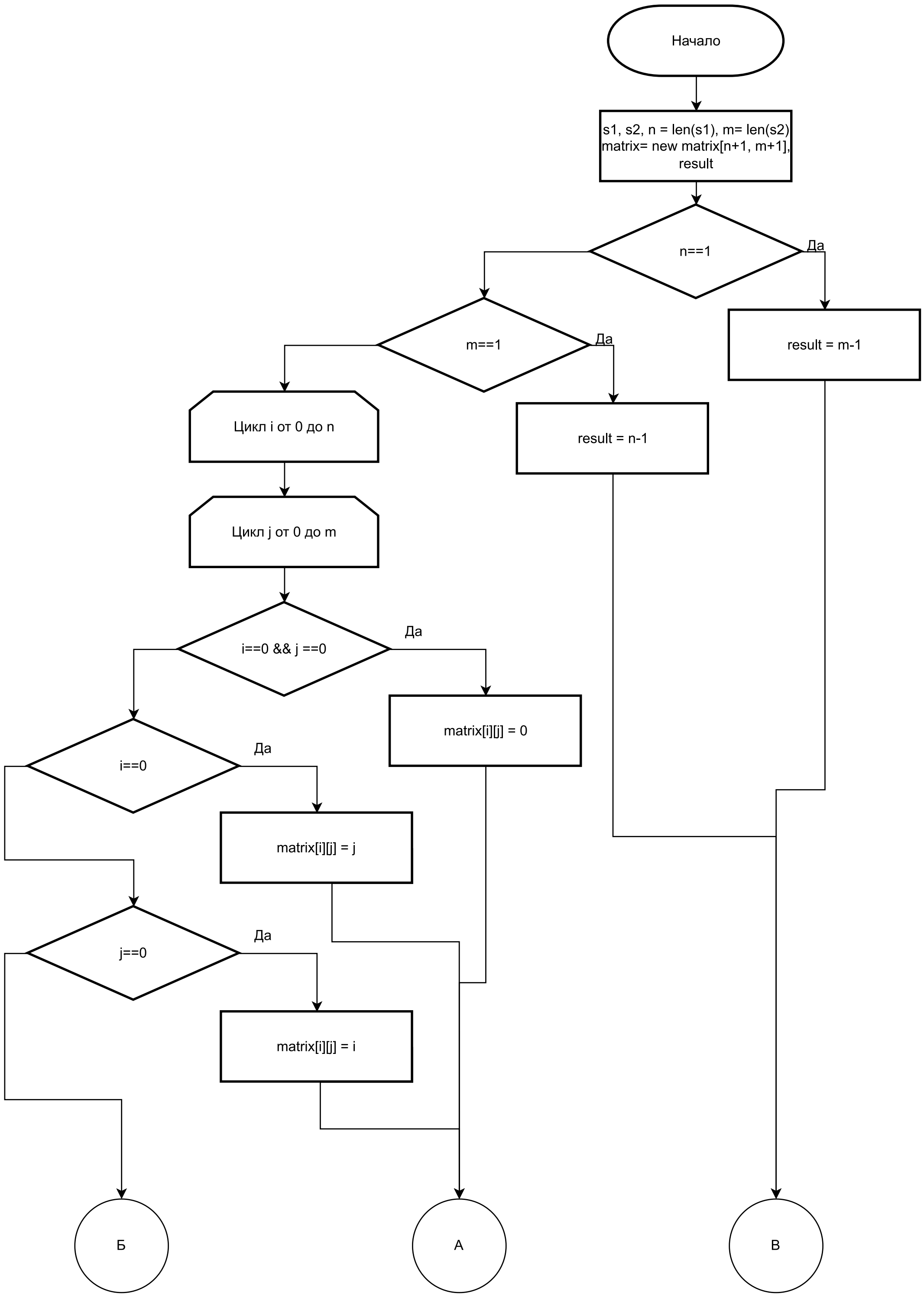

Рис. 2.1 – Схема алгоритма поиска р. Л. матричным способом, часть 1

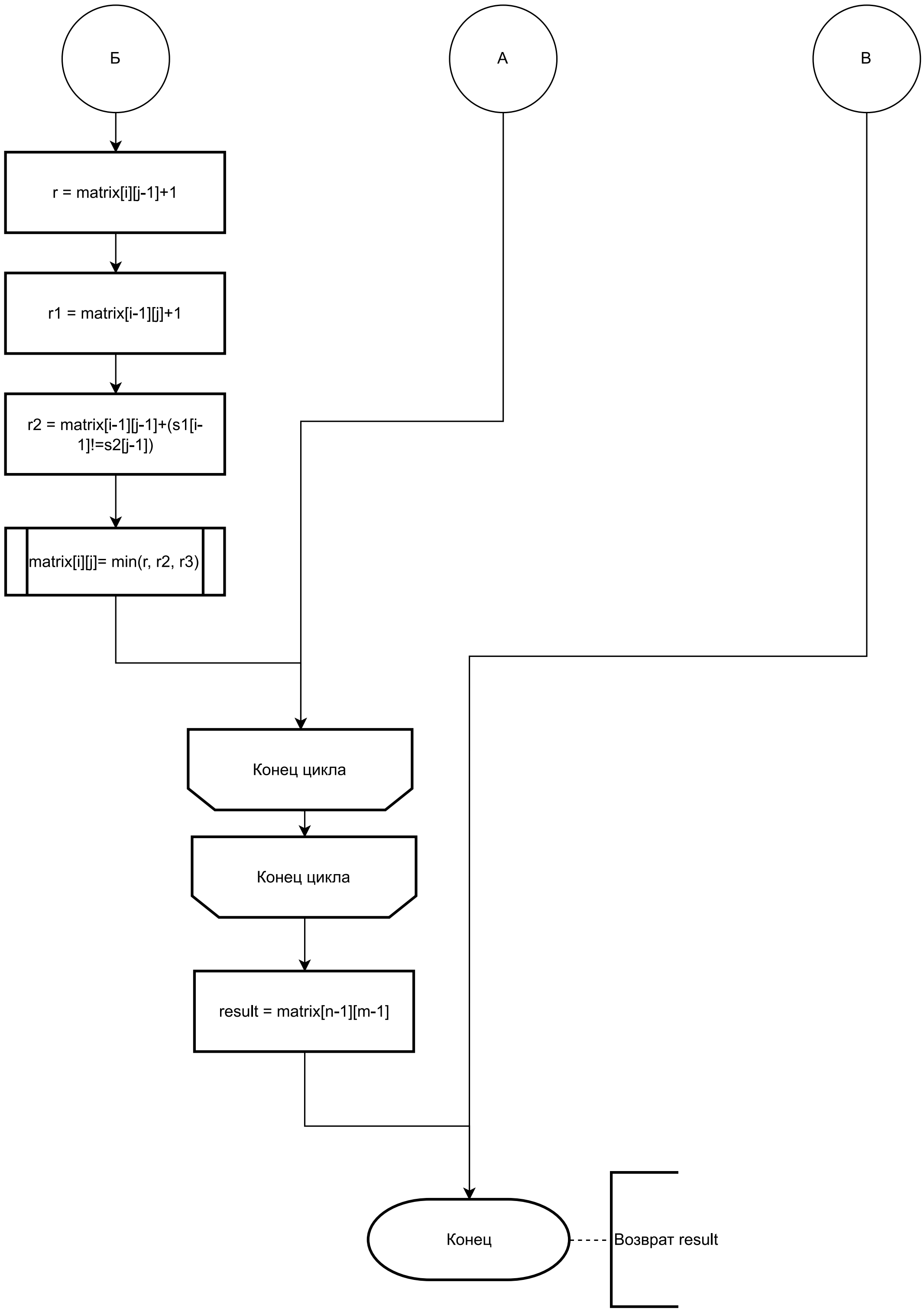

Рис. 2.2 – Схема алгоритма поиска р. Л. матричным способом, часть 2

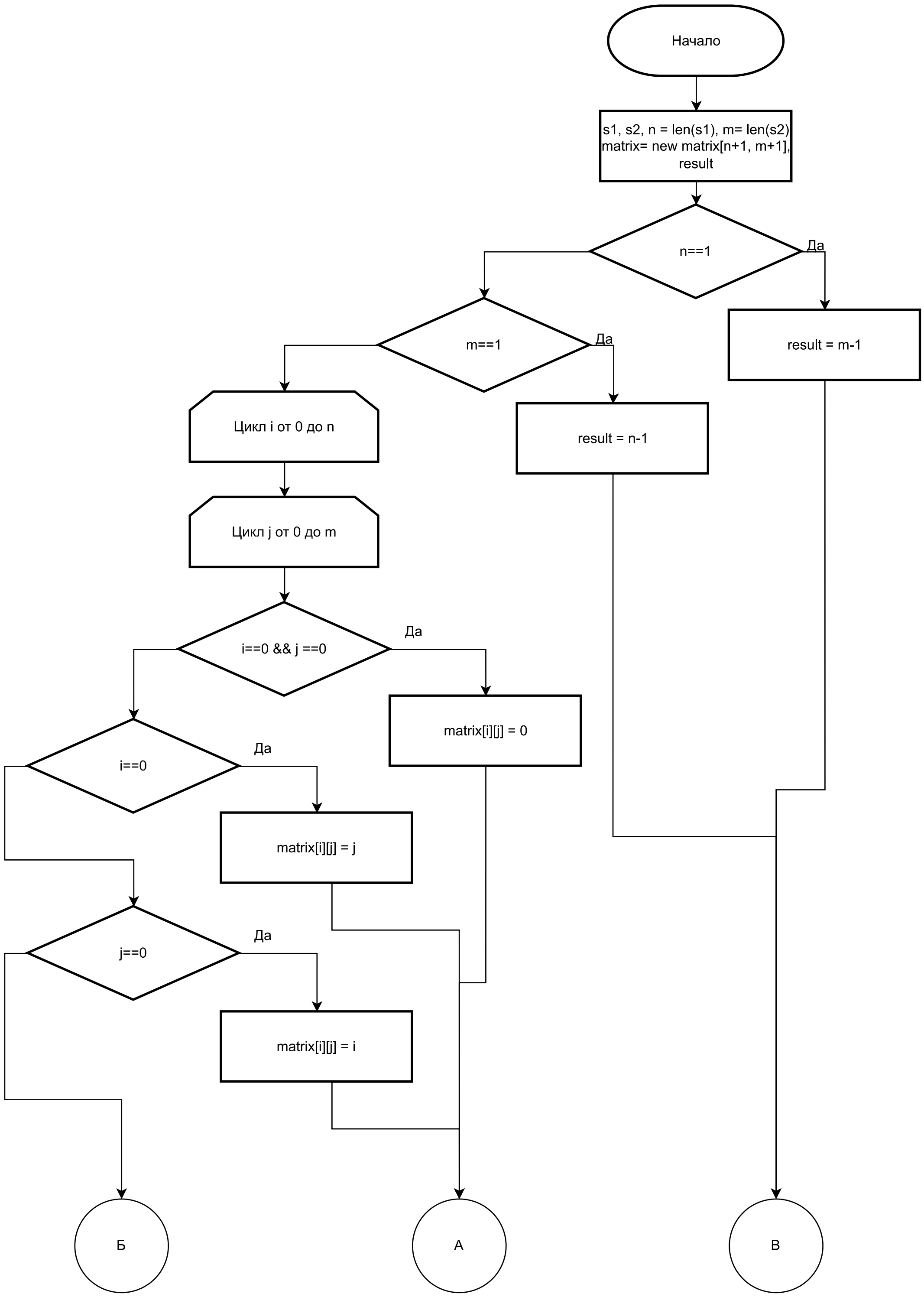

Рис. 2.3 – Схема алгоритма поиска р. Д-Л. матричным способом, часть 1

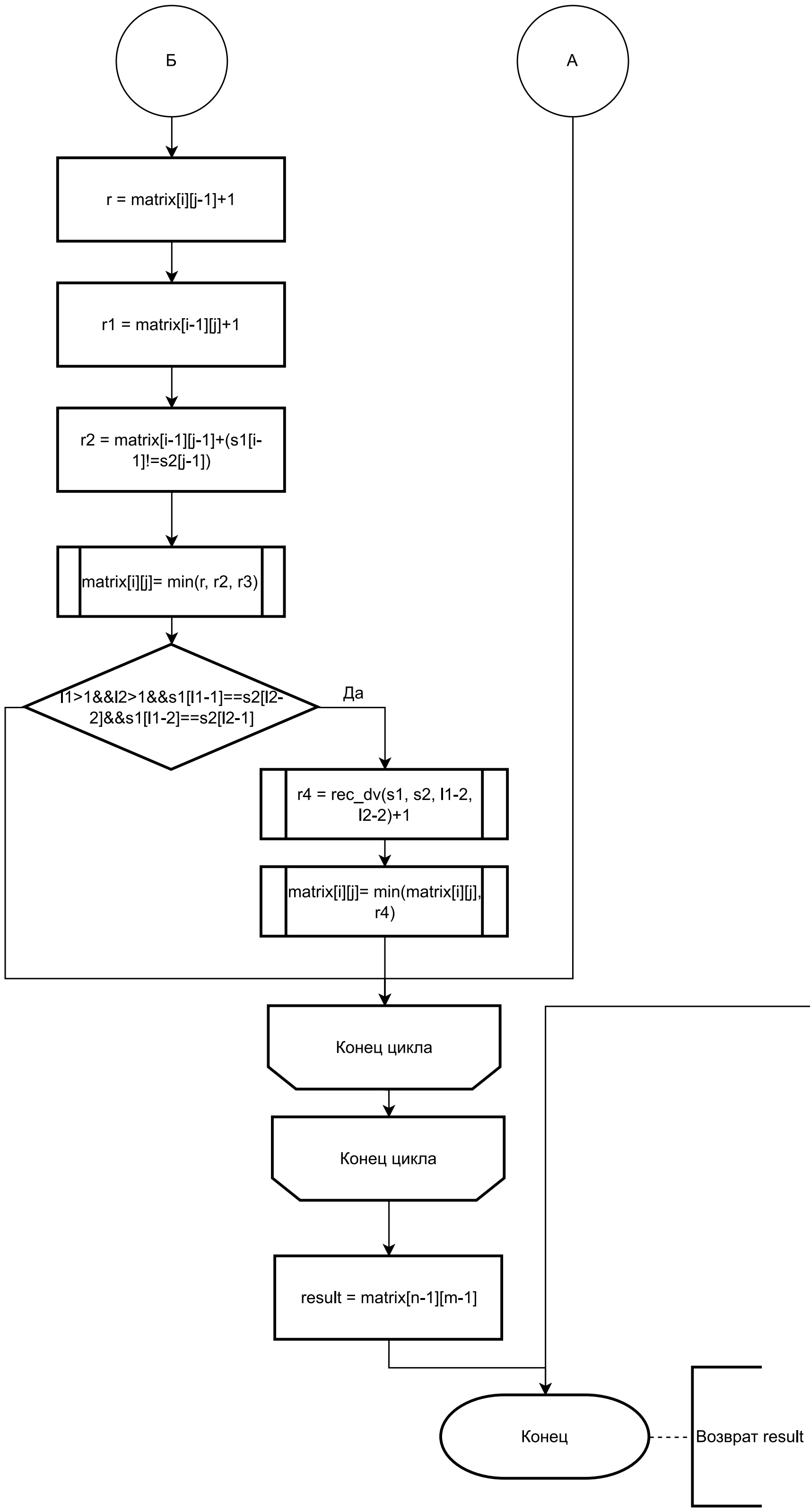

Рис. 2.4 – Схема алгоритма поиска р. Д-Л. матричным способом, часть 2

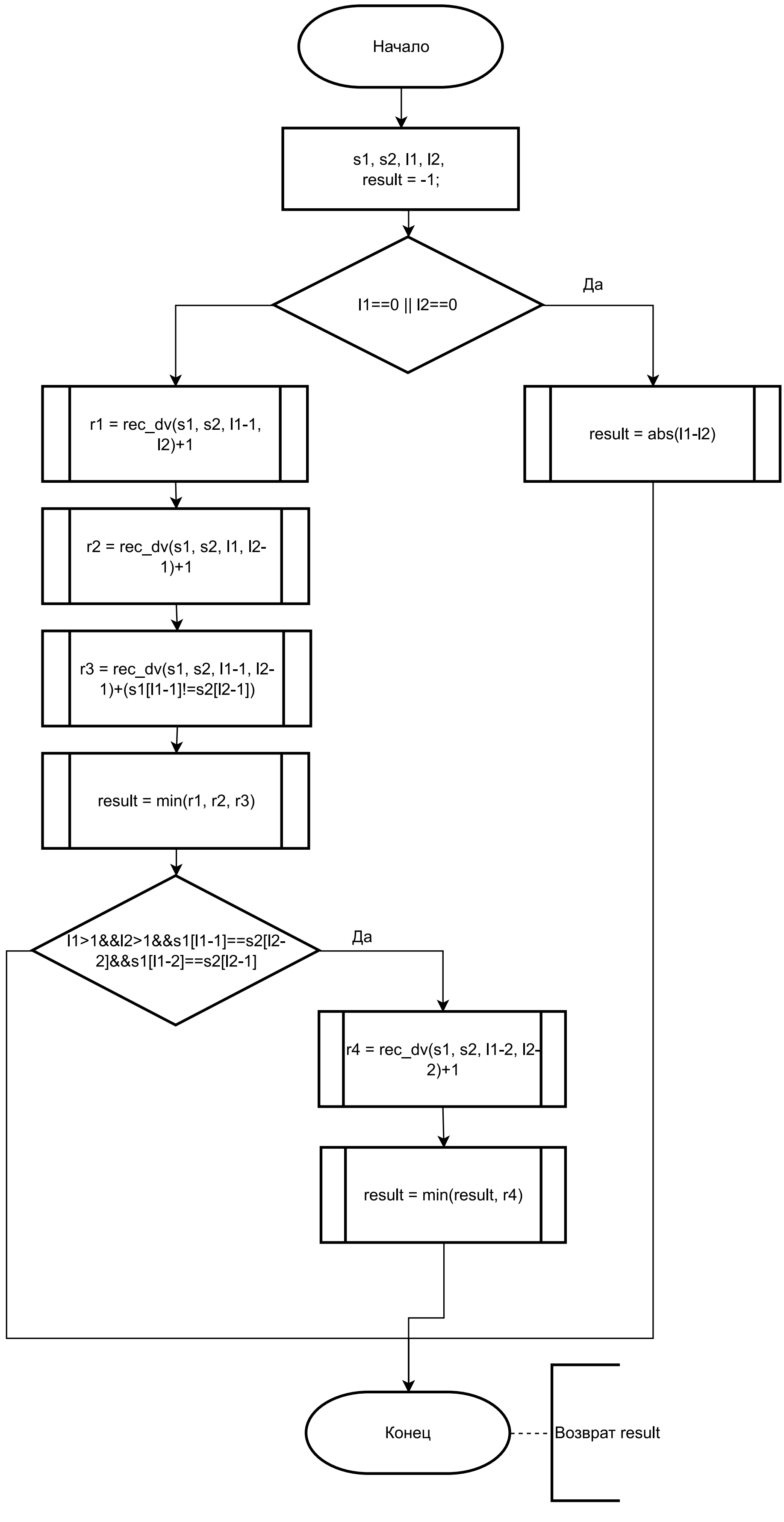

Рис. 2.5 – Схема алгоритма поиска р. Д-Л. рекурсивным способом

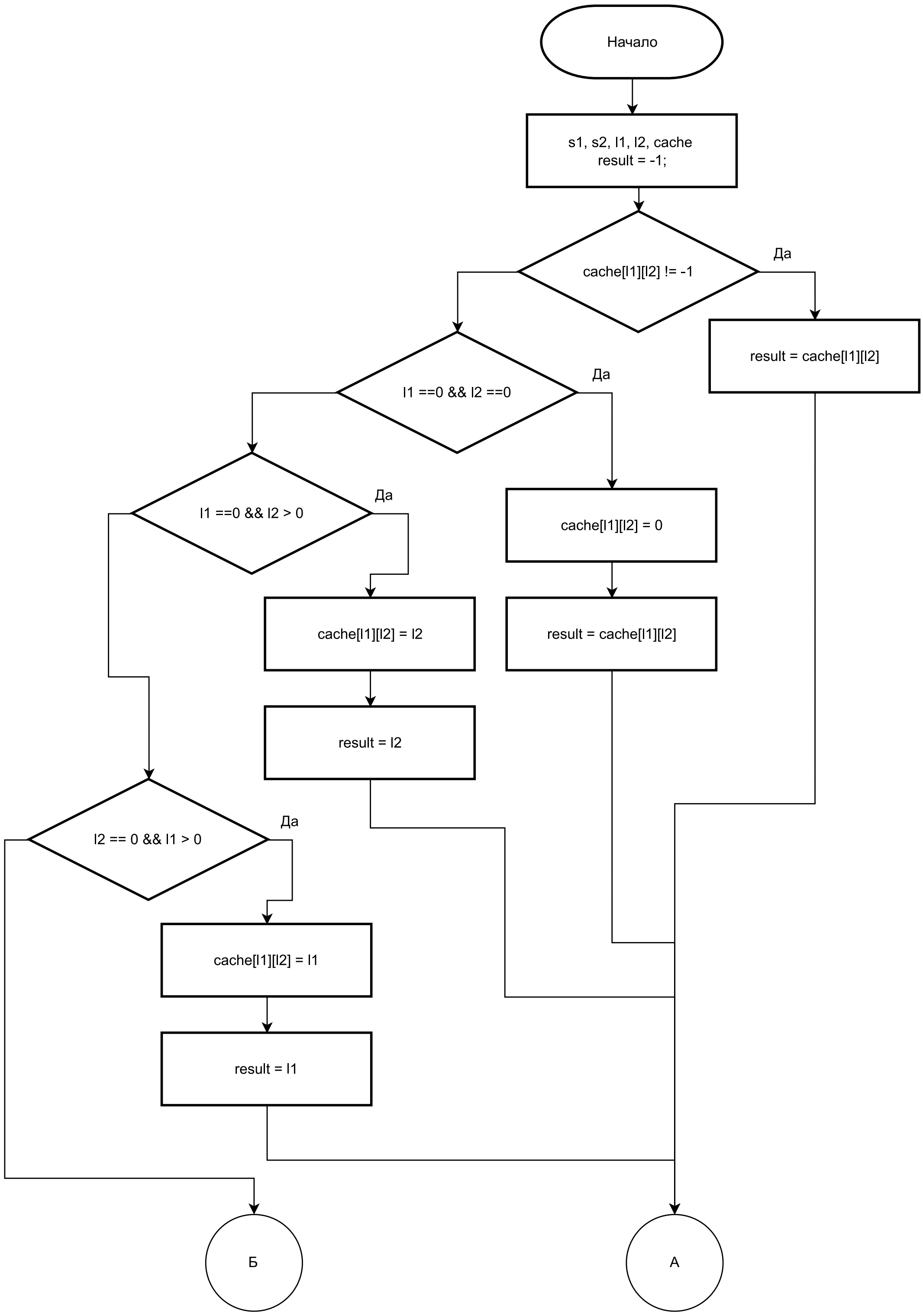

Рис. 2.6 – Схема алгоритма поиска р. Д-Л. рекурсивным способом с кешем, часть 1

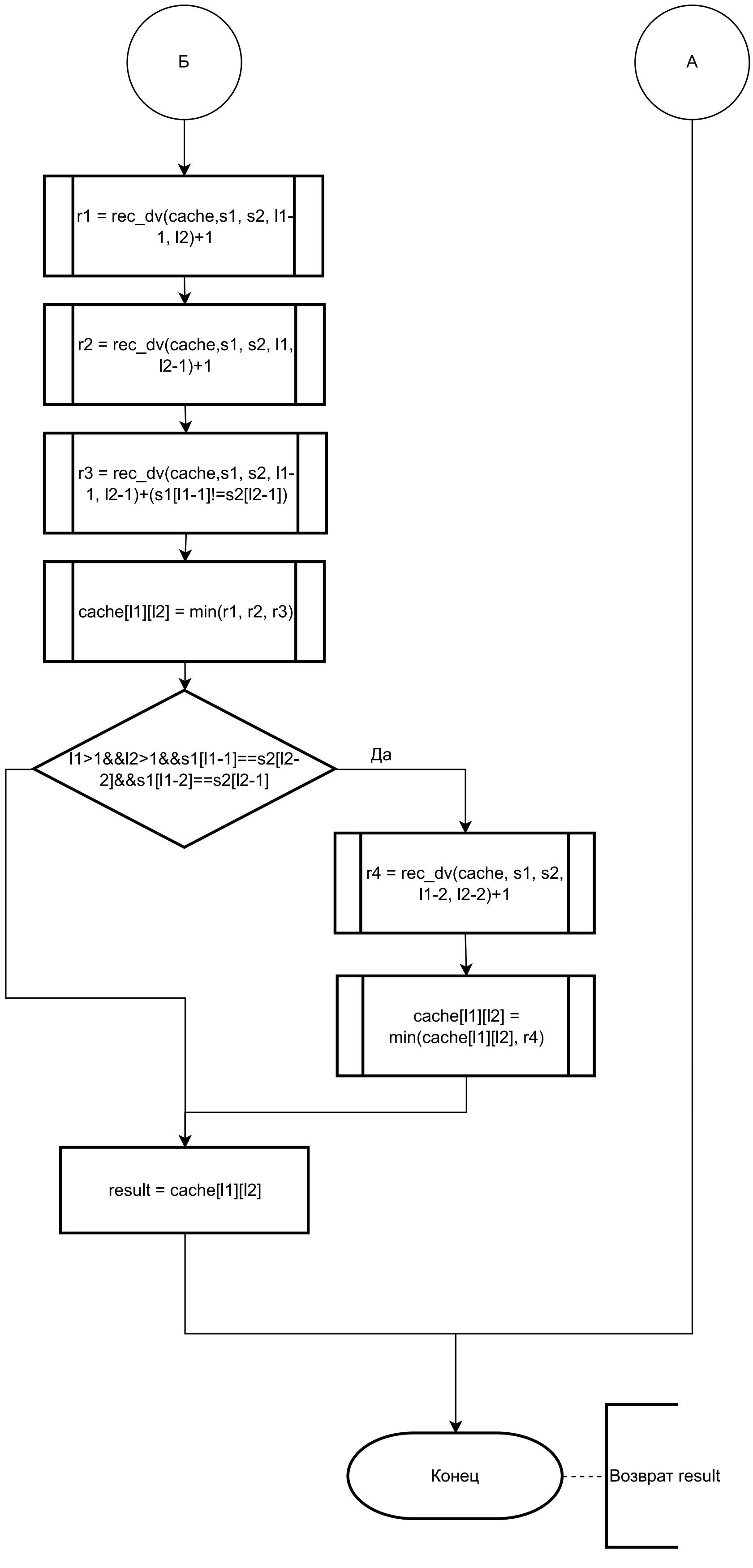

Рис. 2.7 – Схема алгоритма поиска р. Д-Л. рекурсивным способом с кешем, часть 2

2.2 Использование памяти

Замеры времени работы и используемой памяти алгоритмов Левенштейна и Дамерау-Левенштейна могут быть произведены одним и тем же способом. Тогда рассмотрим только рекурсивную и матричную реализации данных алгоритмов.

Пусть n — длина строки S1, m — длина строки S2.

Тогда можно рассчитать затраты по памяти.

- Алгоритм нахождения расстояния Левенштейна (матричный):
  - для матрицы — ((n + 1)· (m + 1)) · sizeof(int));
  - для S1, S2 — (n + m) · sizeof(char);
  - для n, m — 2 · sizeof(int);
  - доп. переменные — 6 · sizeof(int);
  - адрес возврата.
- Алгоритм нахождения расстояния Дамерау-Левенштейна (матричный):
  - для матрицы — ((n + 1) · (m + 1)) · sizeof(int));
  - для S1, S2 — (n + m) · sizeof(char);
  - для n, m — 2 · sizeof(int);
  - доп. переменные — 7 · sizeof(int);
  - адрес возврата.
- Алгоритм нахождения расстояния Дамерау-Левенштейна (рекурсивный), где для каждого вызова:
  - для S1, S2 — (n + m) · sizeof(char);
  - для n, m — 2 · sizeof(int);
  - доп. переменные — 5 · sizeof(int);
  - адрес возврата.

- Алгоритм нахождения расстояния Дамерау-Левенштейна с использованием кеша в виде матрицы (память на саму матрицу: $((n + 1) \cdot (m + 1)) \cdot sizeof(int))$ (рекурсивный), где для каждого вызова:
  - для S1, S2 — $(n + m) \cdot sizeof(char)$;
  - для n, m — $2 \cdot sizeof(int)$;
  - доп. переменные — $5 \cdot sizeof(int)$;
  - указатель на матрицу — sizeof(int**);
  - адрес возврата.

Вывод

В данном разделе — были описаны алгоритма поиска расстояний Левенштейна и Дамерау-Левенштейна, рассчитана паямять, используемая данными алгоритмами.

3 Технологическая часть

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

3.1 Требования к программному обеспечению

Программа должна выводить процессорное время, затрачиваемое каждым из алгоритмов.

3.2 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.

3.3 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 входные строки;
- $Matrix.hpp$ — 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 $Gen.cpp$.

3.4 Реализация алгоритмов

В листингах 3.1 – 3.4, представлены реализации алгоритмов поиска расстояний — Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 — 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)
      mtrx.matrix[i][j] = i;
      else
      mtrx.matrix[i][j] = min_int(3, mtrx.matrix[i][j - 1] + 1,
         mtrx.matrix[i - 1][j] + 1,
      mtrx.matrix[i - 1][j - 1] + (s1[i - 1] != s2[j - 1])
      );
    }
  }
  int result = mtrx.matrix[n - 1][m - 1];
  free_matrix(&mtrx);
  return result;
}
```

Листинг 3.2 — Матричный алгоритм поиска пути Дамерау-Левенштейна

```
int non_rec_dam_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)
      mtrx.matrix[i][j] = i;
      else
      {
        mtrx.matrix[i][j] = min_int(3,
        mtrx.matrix[i][j - 1] + 1,
        mtrx.matrix[i - 1][j] + 1,
        mtrx.matrix[i - 1][j - 1] + (s1[i - 1] != s2[j - 1])
        );
        if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2]
            == s2[j - 1])
        mtrx.matrix[i][j] = min_int(2, mtrx.matrix[i][j], mtrx.
           matrix[i - 2][j - 2] + 1);
      }
    }
  }
  int result = mtrx.matrix[n - 1][m - 1];
  free_matrix(&mtrx);
  return result;
}
```

Листинг 3.3 — Рекурсивый алгоритм поиска пути Дамерау-Левенштейна

```
int rec_dam_lev(const char* s1, const char* s2)
{
  return rec_dam_lev1(s1, s2, strlen(s1), strlen(s2));
}

int rec_dam_lev1(const char* s1, const char* s2, const int li1,
   const int li2)
{
  int result = -1;
  if (li1 == 0 || li2 == 0)
  return std::abs(li1 - li2);
  result = min_int(3,
  rec_dam_lev1(s1, s2, li1 - 1, li2) + 1,rec_dam_lev1(s1, s2, li1
```

```
    , li2 - 1) + 1,
  rec_dam_lev1(s1, s2, li1 - 1, li2 - 1) + (s1[li1 - 1] != s2[li2
    - 1]));
  if (li1 > 1 && li2 > 1 && s1[li1 - 1] == s2[li2 - 2] && s1[li1
    - 2] == s2[li2 - 1])
  result = min_int(2,result,rec_dam_lev1(s1, s2, li1 - 2, li2 -
    2) + 1);
  return result;
}
```

Листинг 3.4 — Рекурсивый с кешем алгоритм поиска пути Дамерау-Левенштейна

```
int cache_dam_lev(const char* s1, const char* s2)
{
  data_t cache;
  cache.matrix = create_matrix(strlen(s1) + 1, strlen(s2) + 1);
  cache.n = strlen(s1) + 1;
  cache.m = strlen(s2) + 1;
  fill_mtrx_with_inf(&cache);
  int r = cache_dam_lev1(&cache, s1, s2, strlen(s1), strlen(s2)
    );
  free_matrix(&cache);
  return r;
}

int cache_dam_lev1(data_t* cache, const char* s1, const char*
  s2, const int li1, const int li2)
{
  if (cache->matrix[li1][li2] != LONG_MAX)
  return cache->matrix[li1][li2];
  if (li1 == 0 && li2 == 0)
  {
    cache->matrix[li1][li2] = 0;
    return cache->matrix[li1][li2];
  }
  if (li1 == 0 && li2 > 0)
  {
    cache->matrix[li1][li2] = li2;
    return li2;
```

```
    }
    if (li2 == 0 && li1 > 0)
    {
      cache->matrix[li1][li2] = li1;
      return li1;
    }
    int r1 = 0, r2 = 0, r3 = 0;
    r1 = cache_dam_lev1(cache, s1, s2, li1 - 1, li2) + 1;
    r2 = cache_dam_lev1(cache, s1, s2, li1, li2 - 1) + 1;
    r3 = cache_dam_lev1(cache, s1, s2, li1 - 1, li2 - 1) + (s1[
       li1 - 1] != s2[li2 - 1]);
    cache->matrix[li1][li2] = min_int(3, r1, r2, r3);
    int result = 0;
    if (li1 > 1 && li2 > 1 && s1[li1 - 1] == s2[li2 - 2] && s1[
       li1 - 2] == s2[li2 - 1])
    {
      int r4 = 0;
      r4 = cache_dam_lev1(cache, s1, s2, li1 - 2, li2 - 2) + 1;
      cache->matrix[li1][li2] = min_int(2, cache->matrix[li1][li2
         ], r4);
    }
    return cache->matrix[li1][li2];
}
```

3.5 Функциональное тестирование

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы поиска пути Левенштейна и Дамерау-Левенштейна. Тесты для всех реализаций алгоритмов пройдены успешно.

Таблица 3.1 — 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 поиска расстояний: Левенштейна матричного, Дамерау-Левенштейна матричного, Дамерау-Левенштейна рекурсивного, Дамерау-Левенштейна рекурсивного с кешерованием. Выполнено тестирование реализаций алгоритмов.

4 Исследовательская часть

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

4.1 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.

4.2 Демонстрация работы программы

На рисунке 4.1 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.

Time in ms

Size	NRL	NRDL	CDL	RDL
1	0.0003125	0.0004375	0.000265625	0
2	0.000703125	0.000640625	0.00071875	0
3	0.000953125	0.0009375	0.000953125	0
4	0.00110937	0.00117187	0.00128125	0.0046875
5	0.00135937	0.00148438	0.0018125	0.0234375
6	0.00179688	0.00184375	0.00225	0.129688
7	0.00198438	0.00223437	0.0024375	0.66875
8	0.00253125	0.002625	0.00332812	3.89062
9	0.003125	0.00310938	0.00403125	21.7312
10	0.00346875	0.00378125	0.0046875	125.583

Рис. 4.1 – Пример работы программы

4.3 Время выполнения реализаций алгоритмов

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

Входные данные: строки размером от 1 до 10 символов для сравнений рекурсивной версии алгоритма поиска расстояния Дамерау-Левенштейна и остальных алгоритмов; строки размером от 10 до 500 символов для остальных случаев сравнения.

Результаты замеров времени работы реализаций алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна на различных входных данных (в мс) приведены в таблицах 4.1, 4.2.

Таблица 4.1 — Процессорное время работы реализаций алгоритмов на малом размере входных строк

Размер	р. Л.(матр.)	р. Д-Л.(матр.)	р. Д-Л.(рек. с кешем)	р. Д-Л.(рек.)
1	0.0003125	0.0004375	0.000265625	0
2	0.000703125	0.000640625	0.00071875	0
3	0.000953125	0.0009375	0.000953125	0
4	0.00110937	0.00117187	0.00128125	0.0046875
5	0.00135937	0.00148438	0.0018125	0.0234375
6	0.00179688	0.00184375	0.00225	0.129688
7	0.00198438	0.00223437	0.0024375	0.66875
8	0.00253125	0.002625	0.00332812	3.89062
9	0.003125	0.00310938	0.00403125	21.7312
10	0.00346875	0.00378125	0.0046875	125.583

Таблица 4.2 — Процессорное время работы реализаций алгоритмов на большом размере входных строк

Размер	р. Л.(матр.)	р. Д-Л.(матр.)	р. Д-Л.(рек. с кешем)
10	0.003125	0	0.00625
20	0.0125	0.00625	0.015625
30	0.00625	0.021875	0.034375
40	0.021875	0.040625	0.028125
50	0.053125	0.065625	0.09375
60	0.084375	0.059375	0.1375
70	0.096875	0.10625	0.19375
80	0.146875	0.15625	0.2125
90	0.184375	0.178125	0.3125
100	0.196875	0.228125	0.4
200	0.85625	0.925	1.58125
300	1.88437	2	3.5
400	3.40937	3.5875	6.34063
500	5.40313	5.59687	9.7375

Также на рисунках 4.2, 4.3, 4.4 приведены графические результаты замеров времени работы алгоритмов в зависимости от линейного размера входных строк.

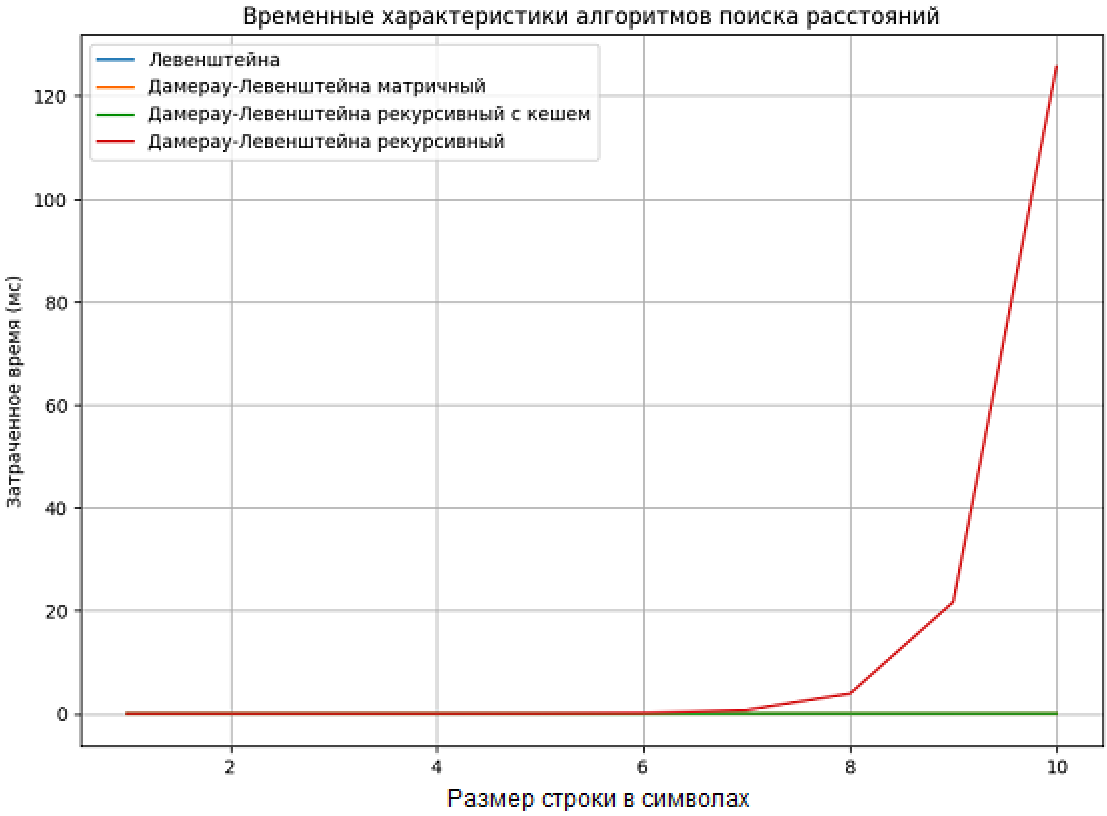

Рис. 4.2 – Процессорное время вычислений

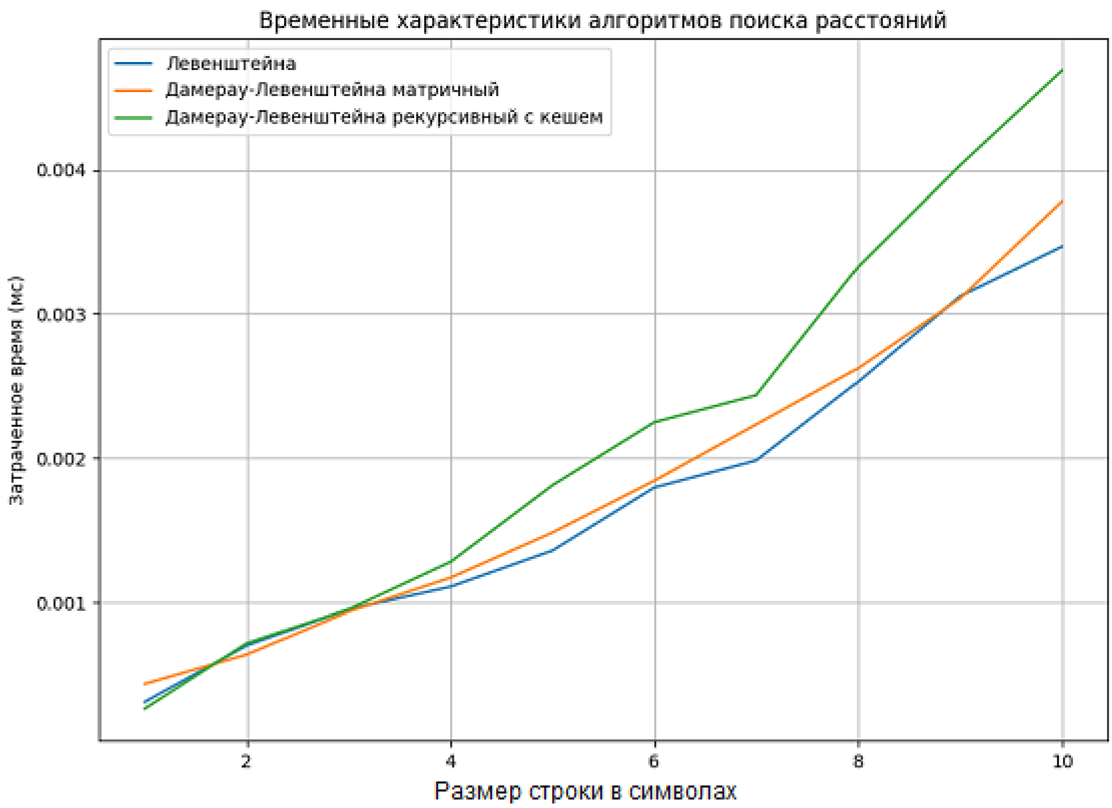

Рис. 4.3 – Процессорное время вычислений

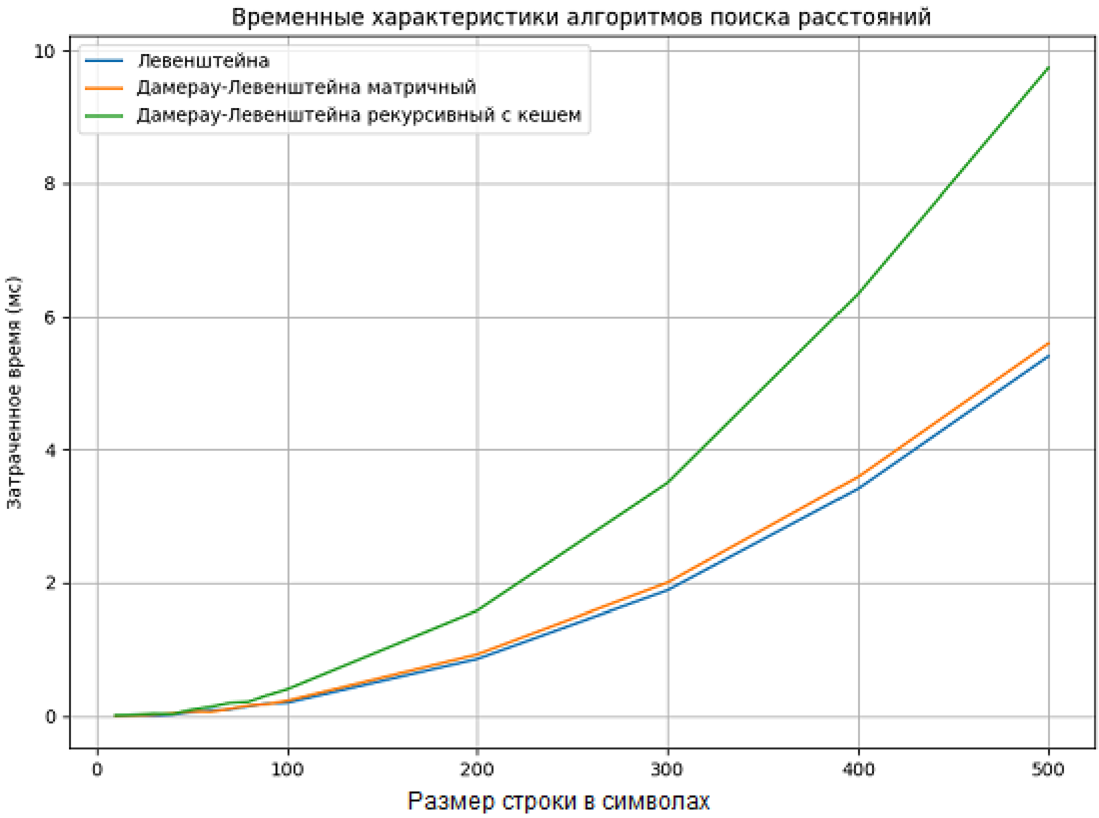

Рис. 4.4 – Процессорное время вычислений

4.4 Вывод

По полученным результатам исследования можно сделать вывод, что рекурсивные реализации поиска расстояний Дамерау-Левенштейна работают гораздо дольше по времени в отличии от матричных реализаций того же алгоритма и алгоритма поиска расстояния Левенштейна: в 41666 и 1.3 раза для рекурсивного и рекурсивного с кешированием соответственно. Таким образом наличие кеша снижает временные затраты на рекурсивный алгоритм в 31000 раз.

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) расчитана затрачиваемая реализованными алгоритмами память;

4) реализован каждый из данных алгоритмов;

5) проведены замеры процессорного времени для каждой из реализаций алгоритмов;

6) выполнен анализ полученных результатов;

7) по итогам работы составлен отчет.

В ходе проделанной работы было выявлено, что реализации рекурсивных алгоритмов поиска расстояний левенштейна и дамерау-Левенштейна требуют больших затрат по времени. Однако рекурсивный алгоритм без использования кеша поиска расстояния Дамерау-Левенштейна требует меньших затрат по памяти.

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).