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 курсу: «Анализ алгоритмов»

Тема: «Расстояние Левенштейна»

Студент: Чыонг Н. В. У.

Группа: ИУ7и-54Б

Преподаватель: Волкова Л. Л.

Оценка: ________________

Москва

2022

Содержание

Введение

Почти каждый день встречается ситуация, когда слово, написанное в поисковике, введено ошибично, и предлагается замена на схожее с ним, или в тектовом редаткоре происходит автозамена ввиду наличия опечатки. Позволить решить эту проблему компьютером позволяет нахождение редакционного расстояния - минимальное количество операций, которые надо совершить, чтобы перевести исходную строку в конечную. [**?**] Благодаря нему можно найти "ближайшее" слово. Одним из базовых видов такого расстояния является *расстояние Левенштейна* (также может использоваться схожее с ним *расстояние Дамерау-Левенштейна*). Помимо этого, оно используется в биоинформатике для определения схожести друг с другом разных участков ДНК или РНК. [1]

Цель работы: получить навык динамического программирования на материале алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна и оценить полученные реализации по памяти и времени. Для достижения цели были поставлены следующие **задачи**:

- изучить алгоритмы нахождения расстояния Левенштейна матричным способом, рекурсивным с ипользованием кэширования и без и расстояния Дамерау-Левенштейна рекурсивным методом;
- разработать алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна перечисленными способами;
- реализовать разработанные алгоритмы;
- провести сравнительный анализ процессорного времени выполнения реализации каждого алгоритма;
- провести сравнительный анализ пиковой затрачиваемой реализациями алгоритмов памяти.

1 Аналитический раздел

В данном разделе рассматриваются различные методы нахождения расстояния Левенштейна (матричный, рекурсивный, рекурсивный с использованием кэширования), рекурсивный способ поиска расстояния Дамерау-Левенштейна.

1.1 Определение

Расстояния Левенштейна, как упоминалось ранее, это базовый вид редакционного расстояния, а точнее - минимальное количество редакторских операций, необходимых для превращения одной строки в другую, - операций вставки (I - insert), удаления (D - delete) и замены (R - replace). [?] Каждая из них имеет цену величиной в 1, и путем посимвольного преобразования необходимо найти такую последовательность операций, чтобы суммарная цена было наименьшей. Для симметричности сравнения еще вводится операция соответствия - match (M). В дальнейшем Ф. Дамерау доказал, что следует добавить еще одну операцию - операцию перестановки двух символов - совокупность этих четырех операций смогут покрыть большинство ошибок при письме, и его способ определения расстояния был назван расстоянием Дамерау-Левенштейна.

1.2 Матричный способ нахождения расстояния Левенштейна

Для поиска расстояния Левенштейна чаще всего используют матрицу D размерами n + 1 и m + 1, где n, m - длины сравниваемых строк s1 и s2. Первая строка и первый столбец заполнются как тривиальные случаи, так как можно однозначно понять, сколько потребуется операций, чтобы превратить пустую строку в строку с одним символом, двумя и т.д. (соответственно одна операция вставки, две и т.д.) и наоборот. Далее каждая ячейка $D_{i,j}$ находится по формуле 1.1.

$$D_{i,j} = min \begin{cases} (D)\ D_{i-1,j} + 1, \\ (I)\ D_{i,j-1} + 1, \\ (R)\ D_{i-1,j-1} + \begin{bmatrix} 1,\ if\ s1[i] == s2[j]; \\ 0,\ else \end{bmatrix} \end{cases} \tag{1.1}$$

Результатом будет являться правая нижняя ячейка в получившейся матрице. Можно заметить, что в выполнении этих действий участвуют только две строки: заполняемая и предыдущая. Поэтому для экономии памяти можно не хранить всю матрицу, а работать только с ними.

Далее приводится пример матрицы 1.2, составленной при сравнении двух строк: КОТ и СКАТ.

$$\begin{pmatrix} \dots & 0 & C & K & A & T \\ 0 & 0 & 1 & 2 & 3 & 4 \\ K & 1 & 1 & 1 & 2 & 3 \\ O & 2 & 2 & 2 & 2 & 3 \\ T & 3 & 3 & 3 & 3 & 2 \end{pmatrix} \tag{1.2}$$

Расстояние Левенштейна равняется двум. Действительно: 1) добавление в начало буквы 'С', 2) замена 'О' на 'А'.

1.3 Рекурсивный способ нахождения расстояния Левенштейна

Рекрсивный способ нахождения расстояния Левенштейна схож с матричным за исключением того, что используется рекурсивная формула

1.3 нахождения результата.

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0,\ if\ i == 0,\ j == 0; \\ i,\ if\ i > 0,\ j == 0; \\ j,\ if\ i == 0, j > 0; \\ min \begin{cases} D(s1[1..i],\ s2[1..j-1) + 1, \\ D(s1[1..i-1],\ s2[1..j]) + 1, \\ D(s1[1..i-1],\ s2[1..j-1]) + \left[\begin{array}{c} 1,\ if\ s1[i] == s2[j], \\ 0,\ else \end{array}\right. \end{cases} \end{cases} \tag{1.3}$$

В функции 1.3:

- если обе строки пустые, то требуется 0 операций;
- если вторая строка пустая, то требуется удалить все символы первой строки;
- если первая строка пустая, то требуется вставить в пустоту все символы второй строки;
- иначе находится минимум среди:
  - суммы расстояния между первой строкой и второй, уменьшенной на 1, и единицы;
  - суммы расстояния между второй строкой и первой, уменьшенной на 1, и единицы;
  - суммы расстояния между первой и второй строками, уменьшенными на 1, и единицы в случае совпадения текущих рассматриваемых символов или нуля иначе.

К существенному недостатку использования данного метода можно отнести нерациональные затраты по времени: сложность алгоритма будет иметь экспоненциальную зависимость, при этом параметры в получающихся функциях могут повторяться, то есть будут повторно пересчитываться уже известные значения.

1.4 Рекурсивный способ нахождения расстояния Левенштейна с использованием кэширования

Решить проблему неэффективного использования рекурсивной формулы нахождения расстояния Левенштейна в виде повторного пересчитывания поможет кэширование. Кэширование - это высокоскоростной уровень хранения, на котором требуемый набор данных временного характера. [?] Благодаря наличию кэша, можно будет подставлять в формулу уже вычисленное ранее значение, если такое имеется. Существует множество способов кэширования, а также уже готовые решения.

1.5 Рекурсивный способ нахождения расстояния Дамерау-Левенштейна

Способ нахождения расстояний Дамерау-Левенштейна и Левенштейна аналогичны. В функции 1.3 в формулу нахождении минимума добавляется еще одна строка 1.4

$$\left[\begin{array}{c} D(s1[1..i-2], s2[1..j-2]) + 1,\ if\ i > 2,\ j > 2,\ s1[i-2] == s2[j-2], \\ \infty,\ else \end{array}\right. \tag{1.4}$$

При невыпонении заданного условия будет присваиваться значение бесконечности, которая заведомо больше любого числа, то есть никак не повлияет на результат.

1.6 Вывод

Таким образом, разобраны способы нахождения расстояний Левенштейна и Дамерау-Левенштейна (отличие последнего состоит в добавлении одного условия), получены формулы. Редакционное расстояние можно получить как матрично, то есть итерационно, так и рекурсивно. Есть возможность сделать эффективней каждый из этих методов:

первый - путем хранения только двух строк матрицы, второй - путем кэширования.

2 Конструкторский раздел

В данном разделе представлены схемы алгоритмов нахождения редакционного расстояния: Левенштейна - с ипользованием матрицы, Левенштейна - рекурсивно, Левенштейна - рекурсивно с использованием кэша, Дамерау-Левенштейна - рекурсивно.

2.1 Алгоритм нахождения расстояния Левенштейна - матрично

На рисунках 2.1 - 2.2 представлена схема алгоритма нахождения расстояния Левенштейна с использованием матрицы.

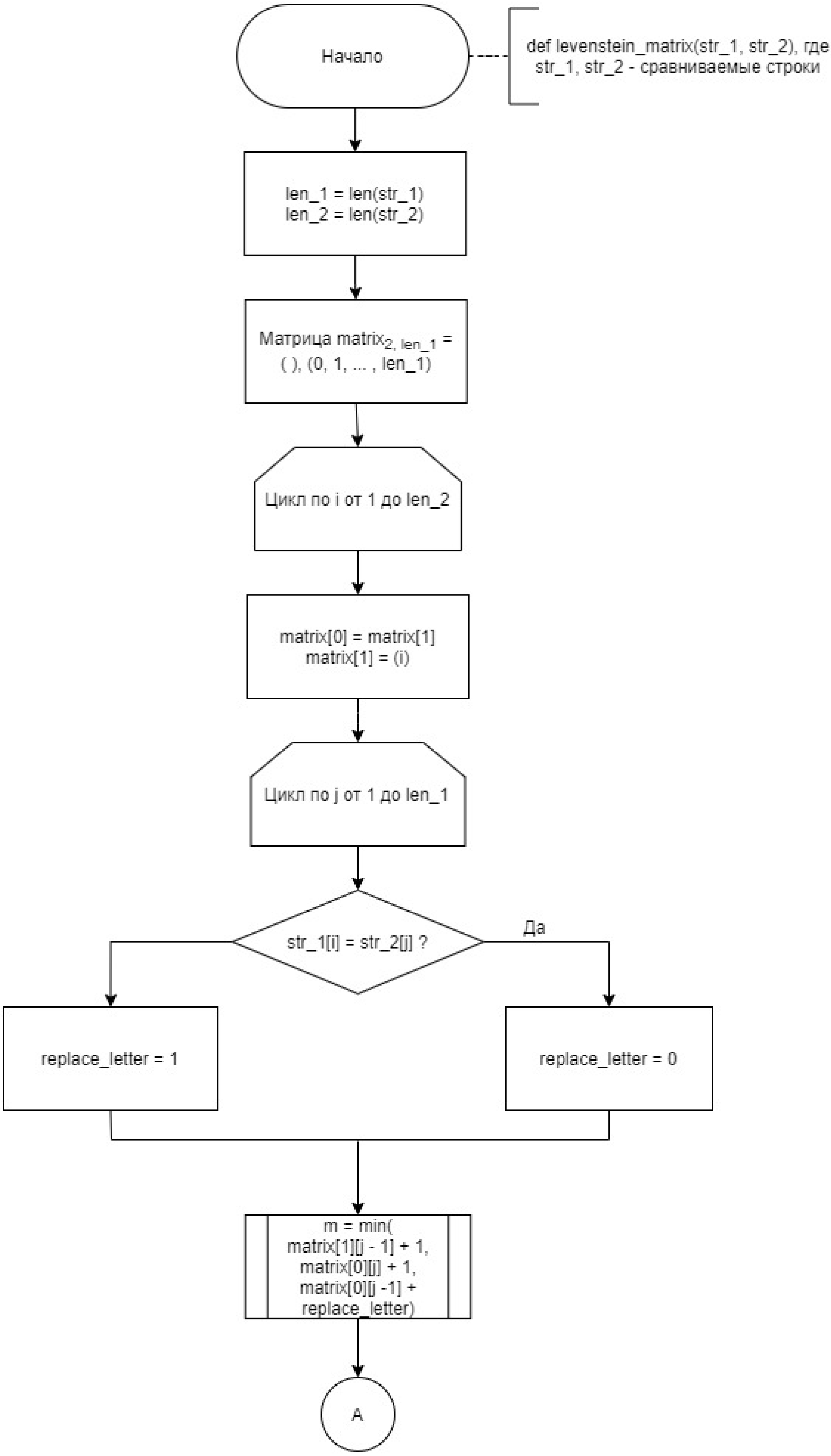

Рис. 2.1: Схема алгоритма нахождения расстояния Левенштейна - матричным способом (часть 1)

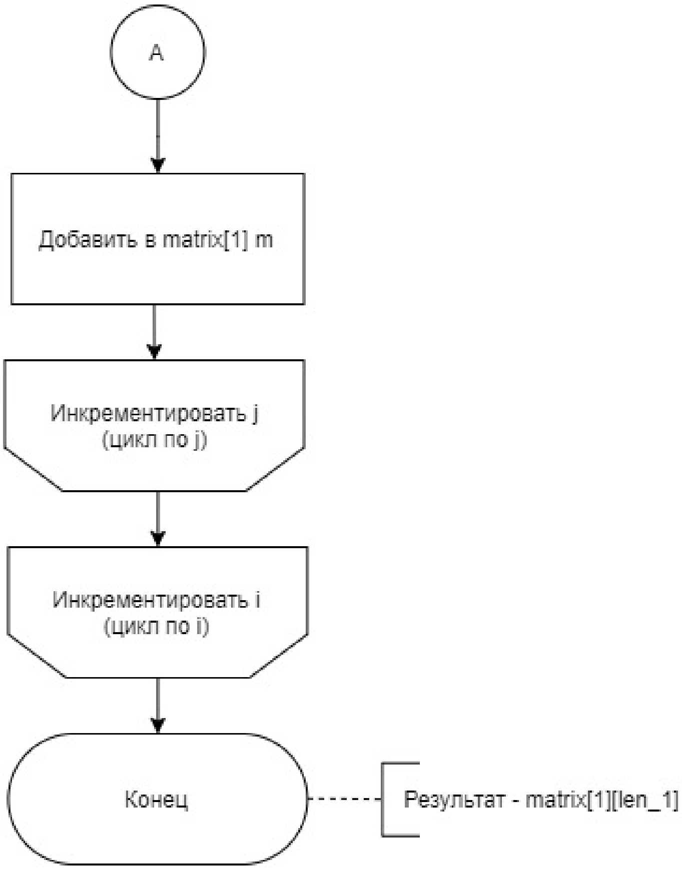

Рис. 2.2: Схема алгоритма нахождения расстояния Левенштейна - матричным способом (часть 2)

2.2 Алгоритм нахождения расстояния Левенштейна - рекурсивно

На рисунках 2.3 - 2.4 представлена схема алгоритма нахождения расстояния Левенштейна рекурсивно.

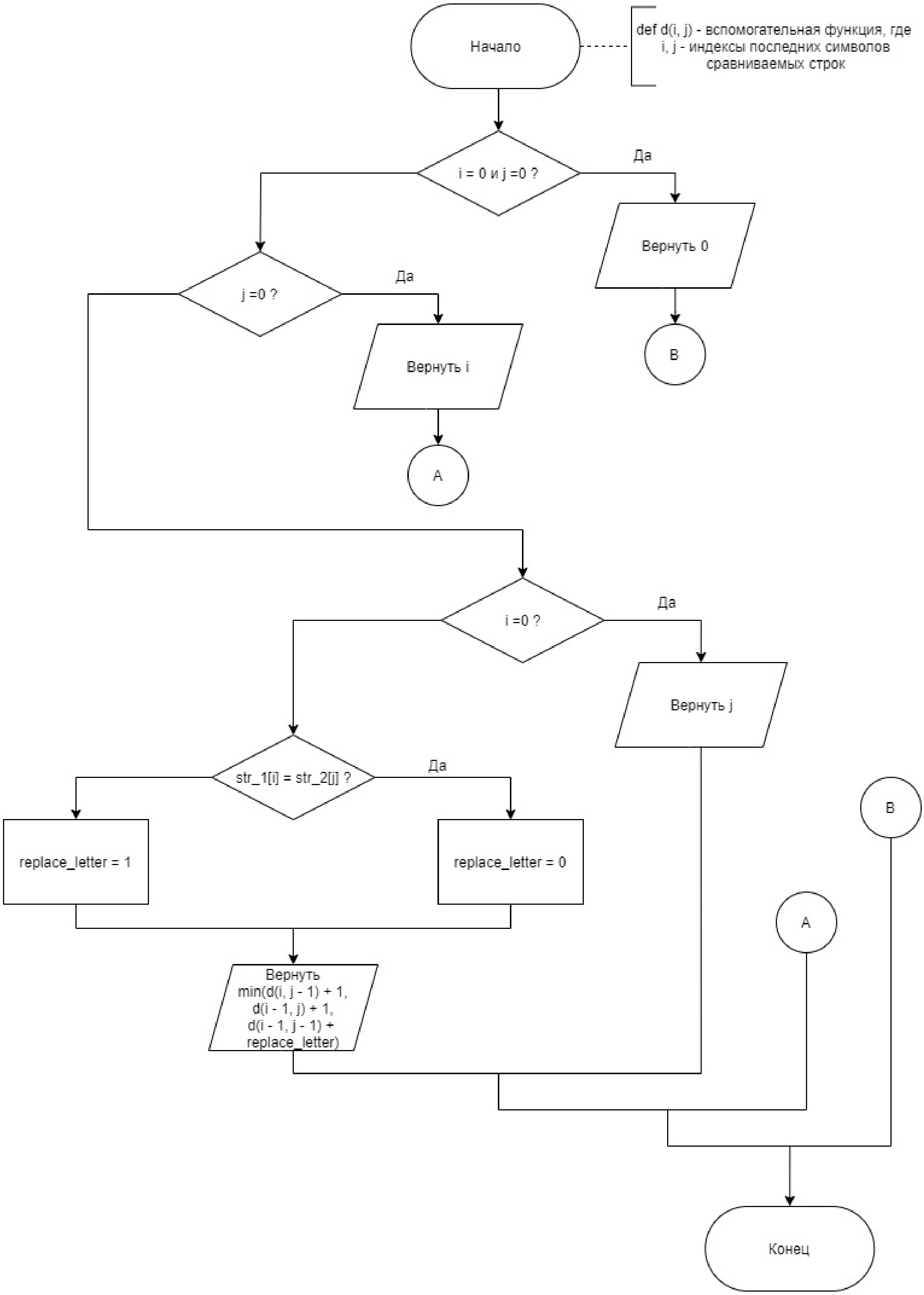

Рис. 2.3: Схема алгоритма нахождения расстояния Левенштейна рекурсивно (часть 1)

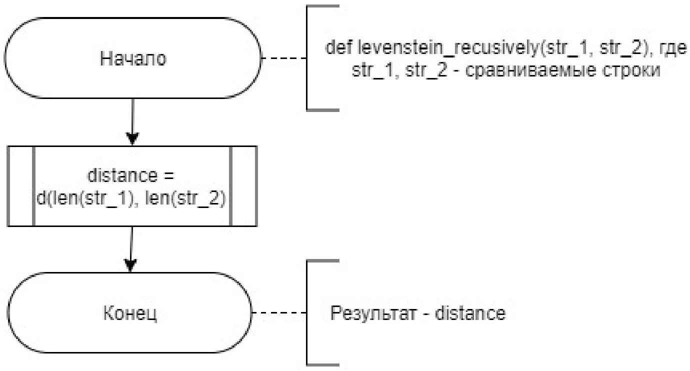

Рис. 2.4: Схема алгоритма нахождения расстояния Левенштейна рекурсивно (часть 2)

2.3 Алгоритм нахождения расстояния Левенштейна - рекурсивно с использованием кэша

На рисунках 2.5 - 2.7 представлена схема алгоритма нахождения расстояния Левенштейна рекурсивно с использованием кэширования.

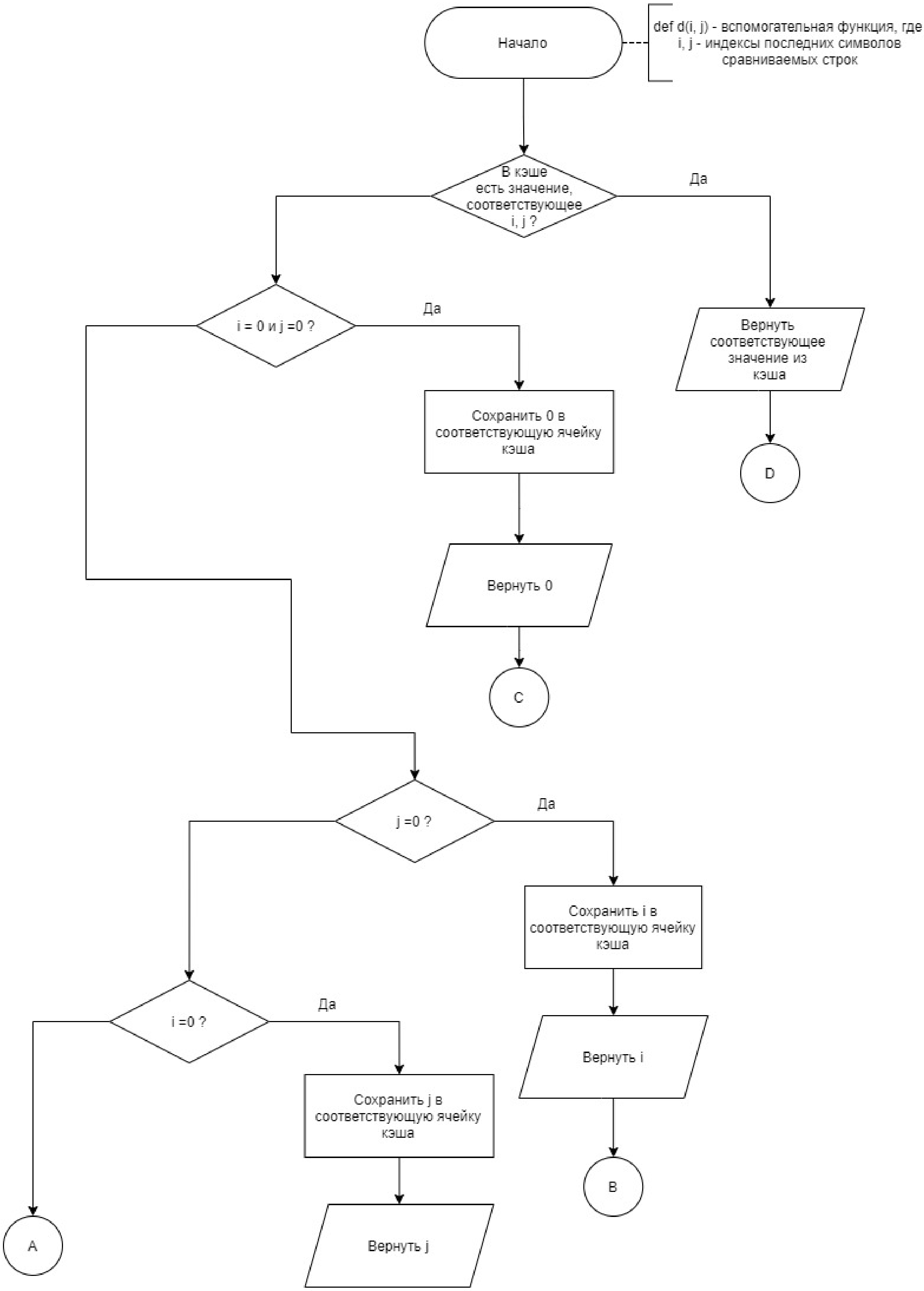

Рис. 2.5: Схема алгоритма нахождения расстояния Левенштейна рекурсивно с использованием кэша (часть 1)

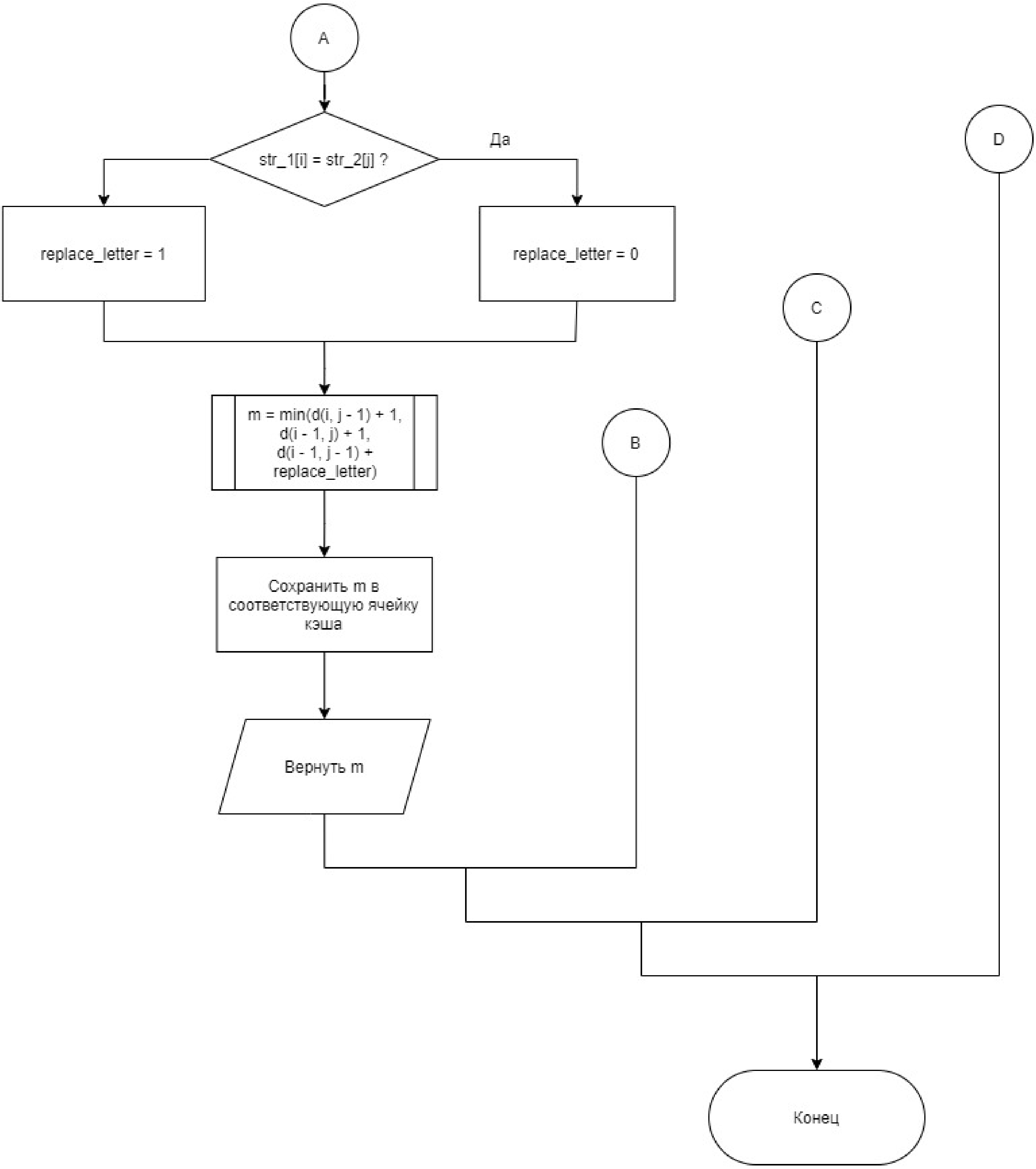

Рис. 2.6: Схема алгоритма нахождения расстояния Левенштейна рекурсивно с использованием кэша (часть 2)

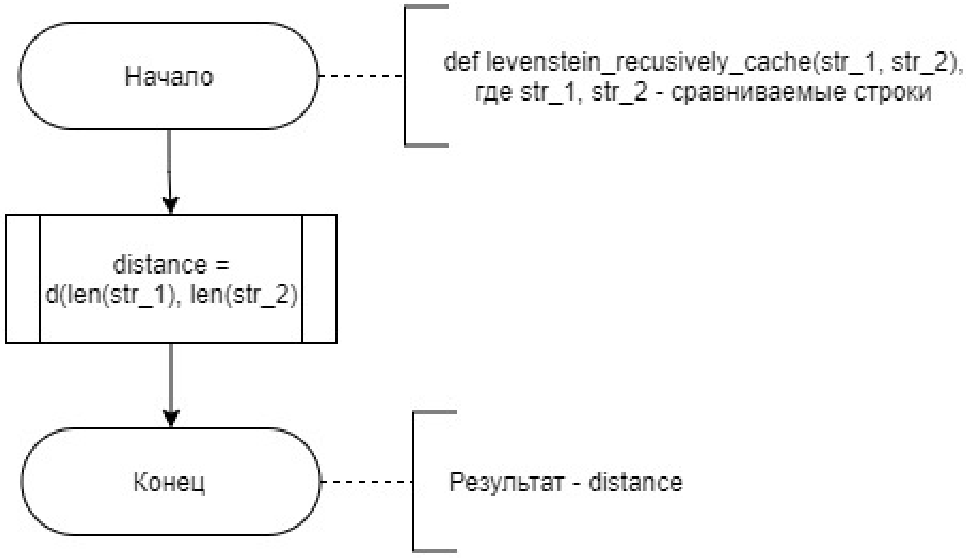

Рис. 2.7: Схема алгоритма нахождения расстояния Левенштейна рекурсивно с использованием кэша (часть 3)

2.4 Алгоритм нахождения расстояния Дамерау-Левенштейна - рекурсивно

На рисунках 2.8 - 2.10 представлена схема алгоритма нахождения расстояния Дамерау-Левенштейна рекурсивно.

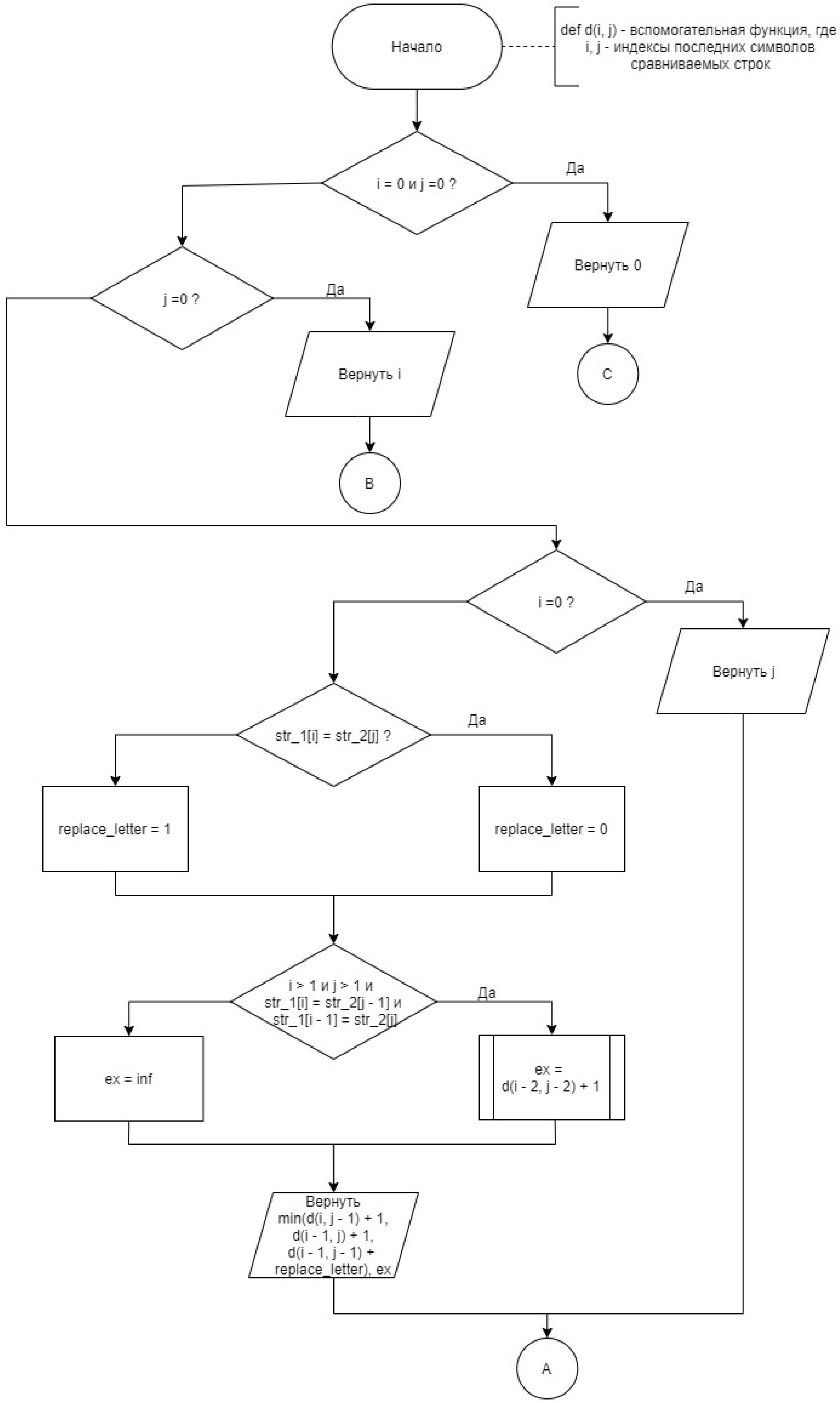

Рис. 2.8: Схема алгоритма нахождения расстояния Дамерау-Левенштейна рекурсивно (часть 1)

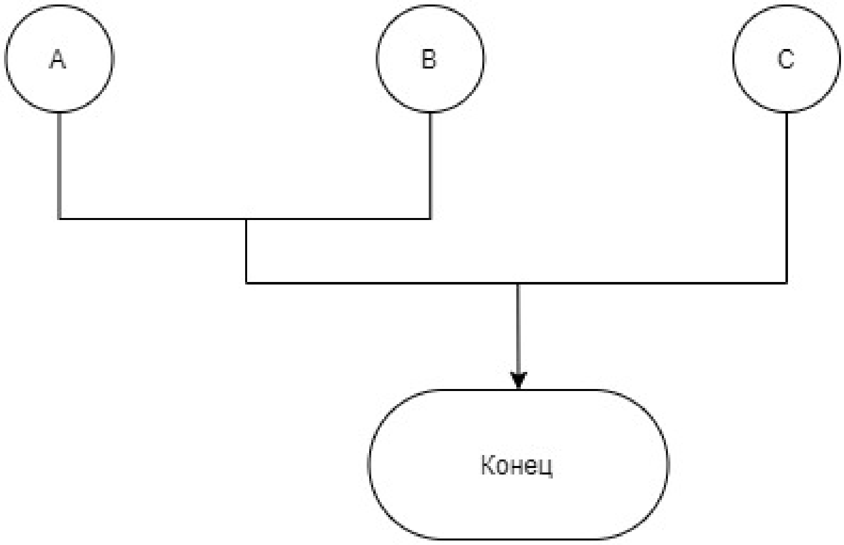

Рис. 2.9: Схема алгоритма нахождения расстояния Дамерау-Левенштейна рекурсивно (часть 2)

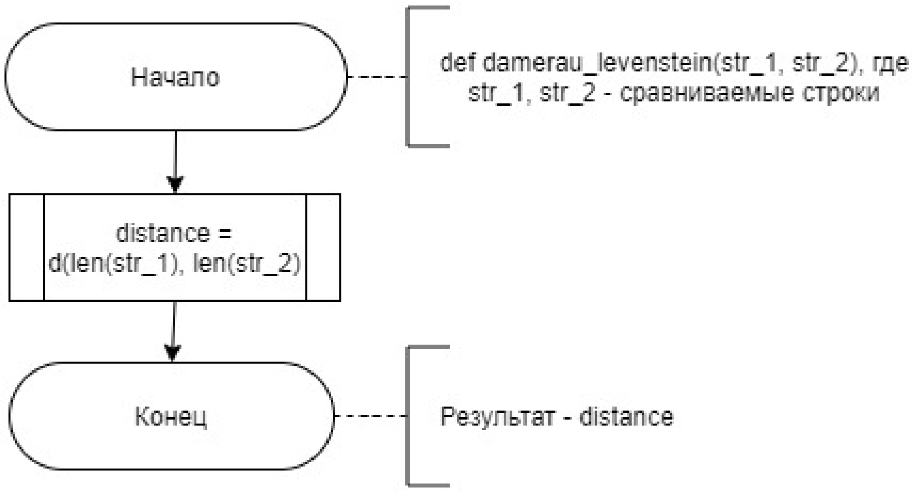

Рис. 2.10: Схема алгоритма нахождения расстояния Дамерау-Левенштейна рекурсивно (часть 3)

2.5 Вывод

Таким образом, были составлены схемы алгоритмов нахождения расстояния Левенштейна тремя разными способами и расстояния Дамерау-Левенштейна - рекурсивно.

3 Технологический раздел

В данном разделе выбирается язык программирования и обоновывается его выбор, вместе с этим подбираются необходимые библиотеки. Также предоставлены листинги реализованных алгоритмов.

3.1 Выбор языка программирования

В качестве языка программирования был выбран язык программирования Python. Благодаря динамической типизации и простому синтаксису, Python позволяет писать быстро и элегантно, позволяя программисту сосредоточиться на реализации самого алгоритма. Также имеется огромное количество библиотек, в том числе и для реализации графического интерфейса, например, PyQt5, на которой выполнен интерфейс реализуемой программы. Для построения графиков используется библиотека matplotlib.

Для реализации кэширования использовалась модуль библиотеки functools - cache (является декоратором, то есть функцией, который принимает в качестве аргумента другую функцию). Также на вход подается максимальный размер кэша. Если вызов функции с данными параметрами уже совершался, то вовзращается значение из кэша. [?]

Для достижения задач, связанных с замером эффективности, выбраны бибилиотеки time (функция process_time_ns() - процессорное время в наносекундах [?]) и tracemalloc (позволяет узнать пиковое значение памяти с момента старта работы функции [?]).

3.2 Реализация алгоритма нахождения расстояния Левенштейна - матрично

На листинге 3.1 предоставлены реализации алгоритмов нахождения расстояния Левенштейна матричным способом.

Листинг 3.1: Реализация алгоритма Левенштейна матричным способом

```
def levenshtein_matrix(str_1, str_2):
  len_1, len_2 = len(str_1), len(str_2)
  matrix = [[], [i for i in range(len_1 + 1)]]

  for i in range(1, len_2 + 1):
    matrix[0], matrix[1] = matrix[1], [i]

    for j in range(1, len_1 + 1):
      replace_letter = 0 if str_2[i - 1] == str_1[j - 1] else 1
      matrix[1].append(
        min(
          matrix[1][j - 1] + 1,
          matrix[0][j] + 1,
          matrix[0][j - 1] + replace_letter
        )
      )

  return matrix[1][len_1]
```

3.3 Реализация алгоритма нахождения расстояния Левенштейна - рекурсивно

На листинге 3.2 предоставлены реализации алгоритмов нахождения расстояния Левенштейна рекурсивно.

Листинг 3.2: Реализация алгоритма Левенштейна рекурсивным способом

```
def levenshtein_recursively(str_1, str_2):
  def d(i, j):
    if i == 0 and j == 0:
      return 0
    elif j == 0:
```

```
      return i
    elif i == 0:
      return j
    else:
      replace_letter = 0 if str_1[i - 1] == str_2[j - 1] else 1
      return min(
        d(i, j - 1) + 1,
        d(i - 1, j) + 1,
        d(i - 1, j - 1) + replace_letter
      )

  distance = d(len(str_1), len(str_2))
  return distance
```

3.4 Реализация алгоритма нахождения расстояния Левенштейна - рекурсивно с использованием кэша

На листинге 3.3 предоставлены реализации алгоритмов нахождения расстояния Левенштейна рекурсивно с использованием кэша.

Листинг 3.3: Реализация алгоритма Левенштейна рекурсивным способом с использованием кэширования

```
def levenshtein_recursively_cache(str_1, str_2):
  @lru_cache(maxsize=len(str_1) * len(str_2))
  def d(i, j):
    if i == 0 and j == 0:
      return 0
    elif j == 0:
      return i
    elif i == 0:
      return j
    else:
      replace_letter = 0 if str_1[i - 1] == str_2[j - 1] else 1
      return min(
        d(i, j - 1) + 1,
        d(i - 1, j) + 1,
        d(i - 1, j - 1) + replace_letter
```

```
    )

  distance = d(len(str_1), len(str_2))
  return distance
```

3.5 Реализация алгоритма нахождения расстояния Дамерау-Левенштейна - рекурсивно

На листинге 3.4 предоставлены реализации алгоритмов нахождения расстояния Дамерау-Левенштейна рекурсивно.

Листинг 3.4: Реализация алгоритма Дамерау-Левенштейна рекурсивным способом

```
def damerau_levenshtein(str_1, str_2):
  def d(i, j):
    if i == 0 and j == 0:
      return 0
    elif j == 0:
      return i
    elif i == 0:
      return j
    else:
      replace_letter = 0 if str_1[i - 1] == str_2[j - 1] else 1
      if i > 1 and j > 1 and str_1[i - 1] == str_2[j - 2] and str_1[i - 2] == str_2[j - 1]:
        exchange = d(i - 2, j - 2) + 1
      else:
        exchange = float('inf')

      return min(
        d(i, j - 1) + 1,
        d(i - 1, j) + 1,
        d(i - 1, j - 1) + replace_letter,
        exchange
      )

  distance = d(len(str_1), len(str_2))
```

```
    return distance
```

3.6 Интерфейс программы

На рисунке 3.1 представлен интерфейс разработанной программы, который пользволяет пользователю ввести две сравниваемые строки и выбрать алгоритм для нахождения редакционного расстояния. Также имеется возможность попарного вывода графиков зависимостей размера строк от используемого алгоритма. Можно вывести максимально используемую память в процессе выполнения каждого алгоритма.

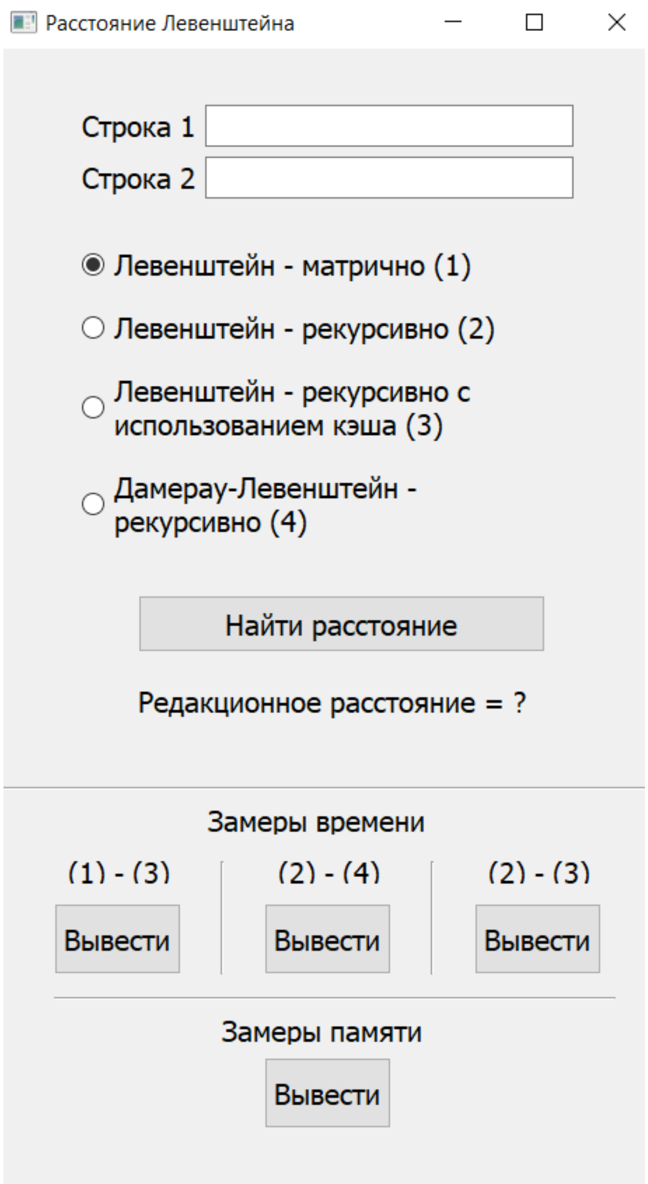

Рис. 3.1: Интерфейс программы

3.7 Вывод

Таким образом, был выбран язык программирования Python для реализации программы, выбраны соответствующие библиотеки, реализованы заданные алгоритмы нахождения расстояний Левенштейна и Дамерау-Левенштейна. Разработан интерфейс, который позволяет пользователю применить каждый из четырех алгоритмов, произвести замеры времени и памяти.

4 Исследовательский раздел

В данном разделе представлены тестовые наборы данных, примеры работы программы, замеры времени и пикового значения памяти для разного размера входных строк и для разных алгоритмов.

4.1 Тестовые наборы

В качестве демонстрации правильности работы алгоритмов протестирован набор данных, представленный в таблице 4.1.

Таблица 4.1: Тестовые наборы к реализованной программе

№	Строка 1	Строка 2	Расстояние Левенштейн	Расстояние Дамерау-Левенштейна	Результат
1			0 - 0 - 0	0	Passed
2	МГТУ		4 - 4 - 4	4	Passed
3		Бауман	6 - 6 - 6	6	Passed
4	скат	скот	1 - 1 - 1	1	Passed
5	рыба	рба	1 - 1 - 1	1	Passed
6	клавиатура	кклавиатура	1 - 1 - 1	1	Passed
7	универ	униевр	2 - 2 - 2	1	Passed
8	агент	ааген	2 - 2 - 2	2	Passed
9	море	моорк	2 - 2 - 2	2	Passed
10	солнце	соллннце	2 - 2 - 2	2	Passed
11	трава	ртаваа	3 - 3 - 3	2	Passed
12	ноутбук	ноубцк	2 - 2 - 2	2	Passed
13	компьютер	кмпьюетр	3 - 3 - 3	2	Passed
14	экран	эан	2 - 2 - 2	2	Passed
15	мышка	мфшак	3 - 3 - 3	2	Passed
16	пиксель	пиесмль	2 - 2 - 2	2	Passed
17	интернет	нитерент	4 - 4 - 4	2	Passed
18	данные	дднneы	3 - 3 - 3	2	Passed
19	диспетчер	дииспечео	3 - 3 - 3	3	Passed
20	алгоритм	агоиртс	4 - 4 - 4	3	Passed

В результате тестрования все тесты пройдены успешно, что показывает правильность работы алгоритма на различных строках.

4.2 Примеры работы программы

На рисунках 4.1 - 4.2 представлены примеры работы программы.

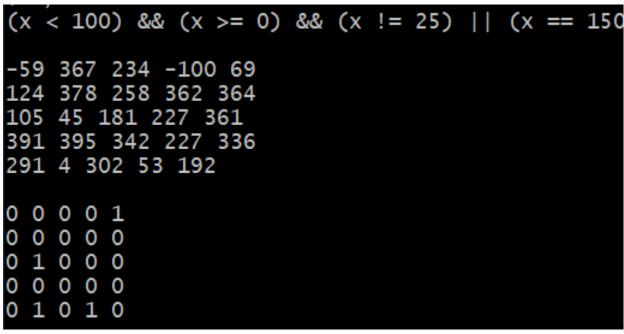

Рис. 4.1: Пример работы №1

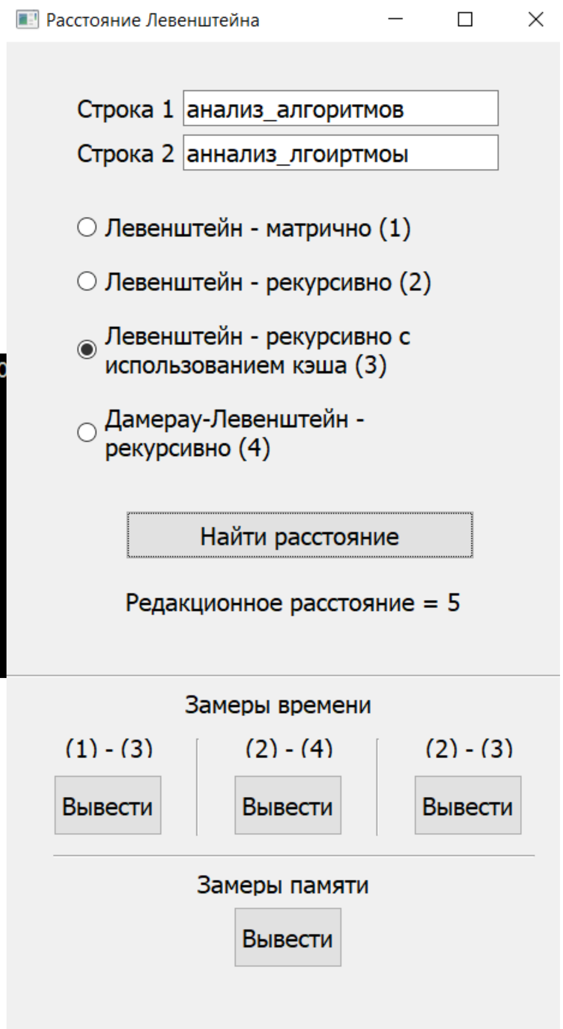

Рис. 4.2: Пример работы №2

4.3 Время выполнения алгоритмов

Как упоминалось ранее для сравнения эффективности алгоритмов используются стандартные библиотеки time и functools.

Разработанная программа предоставляет интерфейс, позволяющий пользователю получить зависимости времени выполнения алгоритма от размера строки для различных пар: 1) Левенштейн (матрично) - Левенштейн (рекурсивно с использованием кэша)); 2) Левенштейн (рекурсивно) - Дамерау-Левенштейн (рекурсивно); 3) Левенштейн (рекурсивно) - Левенштейн (рекурсивно с использованием кэша). Соответствующие графики которых изображены на рисунках 4.3 - 4.5.

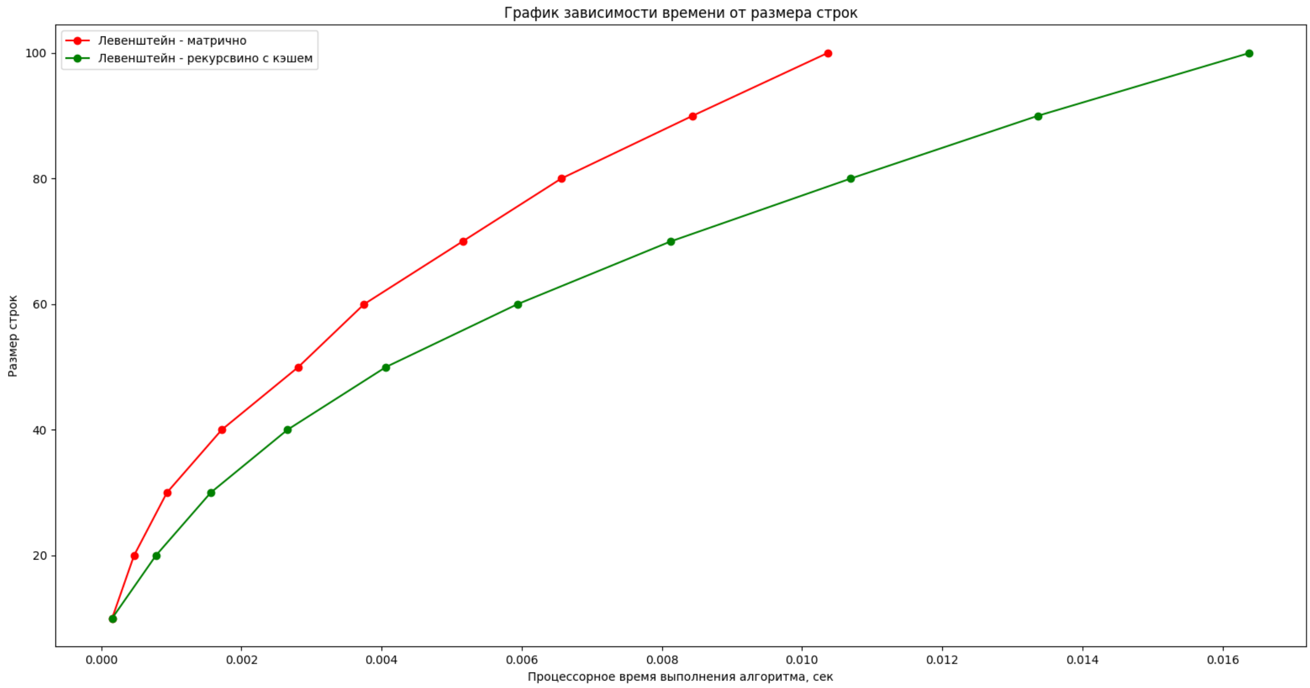

Рис. 4.3: Зависимости времени работы алгоритмов поиска расстояния Левенштейна матричным способом и рекурсивным с использованием кэша от размера строк

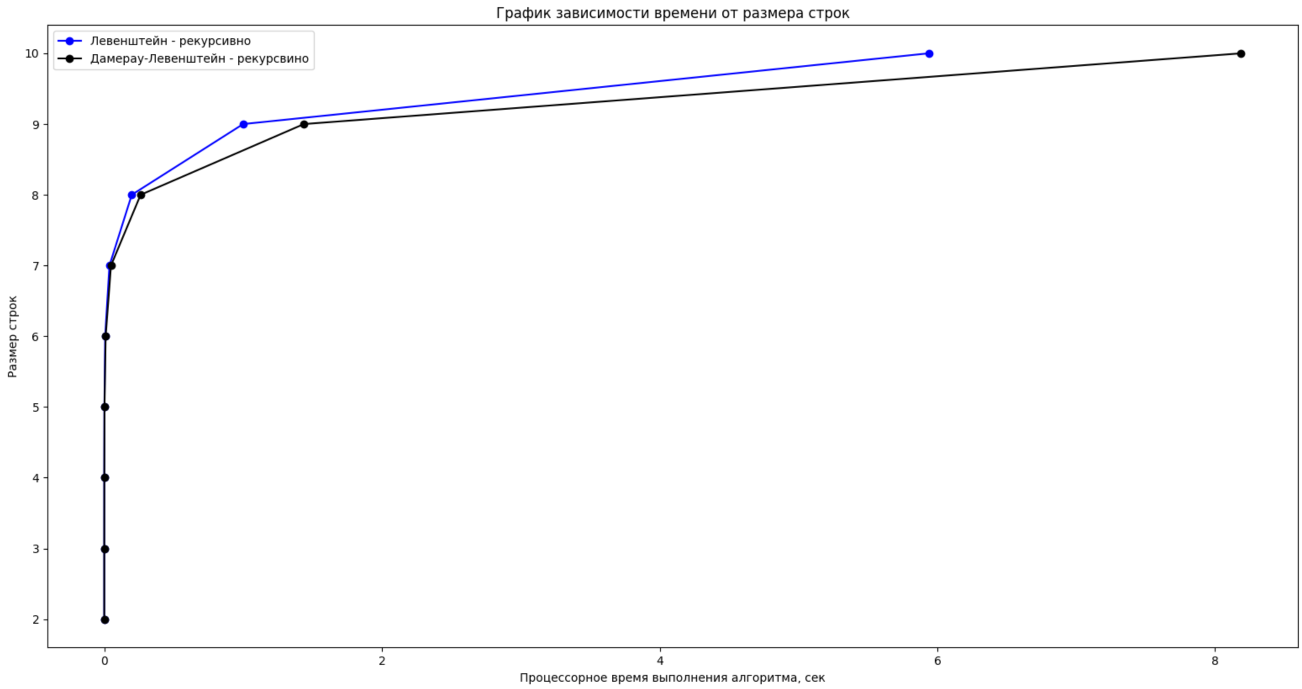

Рис. 4.4: Зависимости времени работы алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна рекурсивно от размера строк

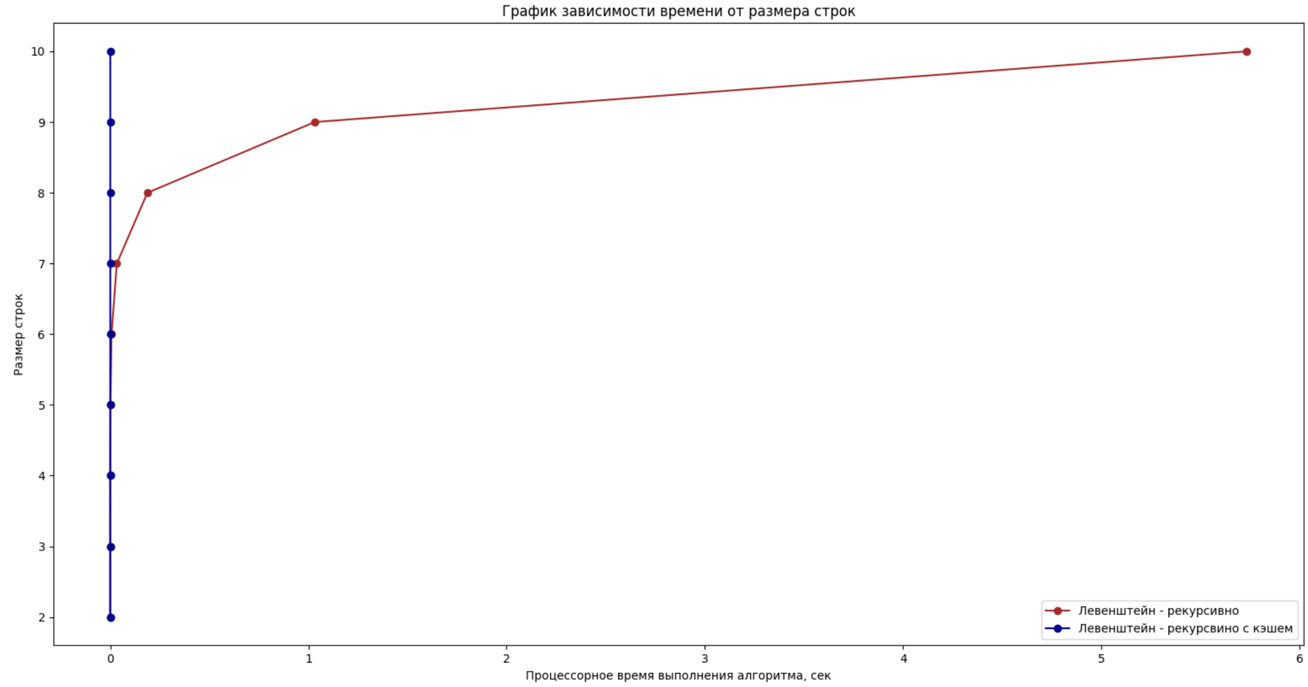

Рис. 4.5: Зависимости времени работы алгоритмов поиска расстояния Левенштейна рекурсивно с использованием кэша и без от размера строк

Из рисунка 4.3 видно, что матричный способ и рекурсивный с использованием кэширования являются достаточно эффективными по времени (∼(0.1 - 0.2) секунды на обработку строк длинами 100), при этом на всех длинах использование матрицы показывает лучший результат (в ∼1.5 раза). Рекурсивные реализации алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна (рисунок 4.4) долго производят обработку строк - время растет геометрически. Ожидаемо, что для расстояния Дамерау-Левенштейна медленней, так как требуется больше действий. Рисунок 4.5 показывает, какое преимущество дает использование кэширования при рекурсивной реализации. Видно, что в таком случае значения колеблются в пределах 0, тогда как для обычной версии при размере строк 10 тратится ∼6 секунд.

4.4 Пиковое значение памяти

На рисунке

4.5 Вывод

Заключение

В ходе выполнения лабораторной работы была проделана следующая работа:

- были теоретически изучены алгоритмы нахождения расстояний Левенштейна и Дамерау–Левенштейна;
- для некоторых реализаций были применены методы динамического программирования, что позволило сделать алгоритмы быстрее;
- были практически реализованы алгоритмы в 2 вариантах: рекурсивном и итеративном;
- на основе полученных в ходе экспериментов данных были сделаны выводы по поводу эффективности всех реализованных алгоритмов;
- был подготовлен отчет по ЛР.

Список использованных источников

[1] В. И. Левенштейн. *Двоичные коды с исправлением выпадений, вставок и замещений символов*, volume 163, chapter 4, pages 845–848. – М.: Доклады АН СССР, 1965.