Государственное образовательное учреждение высшего профессионального образования
“Московский государственный технический университет имени Н.Э.Баумана”

Дисциплина: Анализ алгоритмов

Лабораторная работа №1

ТЕМА-расстояния Левенштейна и Дамерау-Левенштейна.

Студент группы ИУ7-55,
Шестовских Николай Александрович

2019 г.

Оглавление

Введение

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

Расстояние Левенштейна используют:

- для исправления ошибок в слове;
- для сравнения текстовых файлов утилитой diff и ей подобными;
- для сравнения ДНК в биоинформатике.

Цель работы: изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Задачи 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: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4. Сравнительный анализ линейной и рекурсивной реализаций 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 определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
6. Описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

0.1 Аналитическая часть

0.1.1 Описание алгоритмов

Обозначим простейшие действия над двумя строками:

1. D - удалить букву в одной из строк;
2. I - вставить букву в одной из строк;
3. R - заменить букву в одной из строк;
4. M - совпадение двух букв.

Каждая операция имеет свою ‘цену‘ - у совпадения она 0, а у трех остальных - по 1. Задача нахождения расстояния Левенштейна между двумя строками сводится к поиску набора операций, которые нужно совершить, чтобы трансформировать одну строку в другую, причем их суммарная цена должна быть минимальной.

Расстояние Дамерау-Левенштейна отличается от расстояния Левенштейна добавлением операции транспозиции - перестановки двух соседних символов(ее цена - 1).

Пусть S_1 и S_2 — две строки над некоторым алфавитом, тогда расстояние Левенштейна между ними можно подсчитать по следующей формуле (1):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(D1, D2, D3) \end{cases} \quad (1)$$

где:
D1 = D(i,j-1)+1;
D2 = D(i-1, j) + 1;
D3 = D(i-1, j-1) + 0, match($S_1[i]$, $S_2[j]$));
match(a, b) = истина при a = b, ложь иначе;
min(A1, A2, ..., AN) - минимум среди чисел A1, A2, ..., AN.
В свою очередь, расстояние Дамерау-Левенштейна можно посчитать по формуле (2):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(D1, D2, D3, D4), & i > 1, j > 1, a_i = b_{j-1}, a_{i-1} = b_j \\ min(D1, D2, D3) \end{cases} \quad (2)$$

где:
D4 = D(i-2, j-2) + 1, если i>1 ,j>1 и $a_i = b_{j-1}, a_{i-1} = b_j$.

0.1.2 Вывод

В данном разделе были рассмотрены формулы для нахождения расстояния Левенштейна и Дамерау-Левенштейна, а также основные приложения этих расстояний.

0.2 Конструкторская часть

0.2.1 Разработка алгоритмов

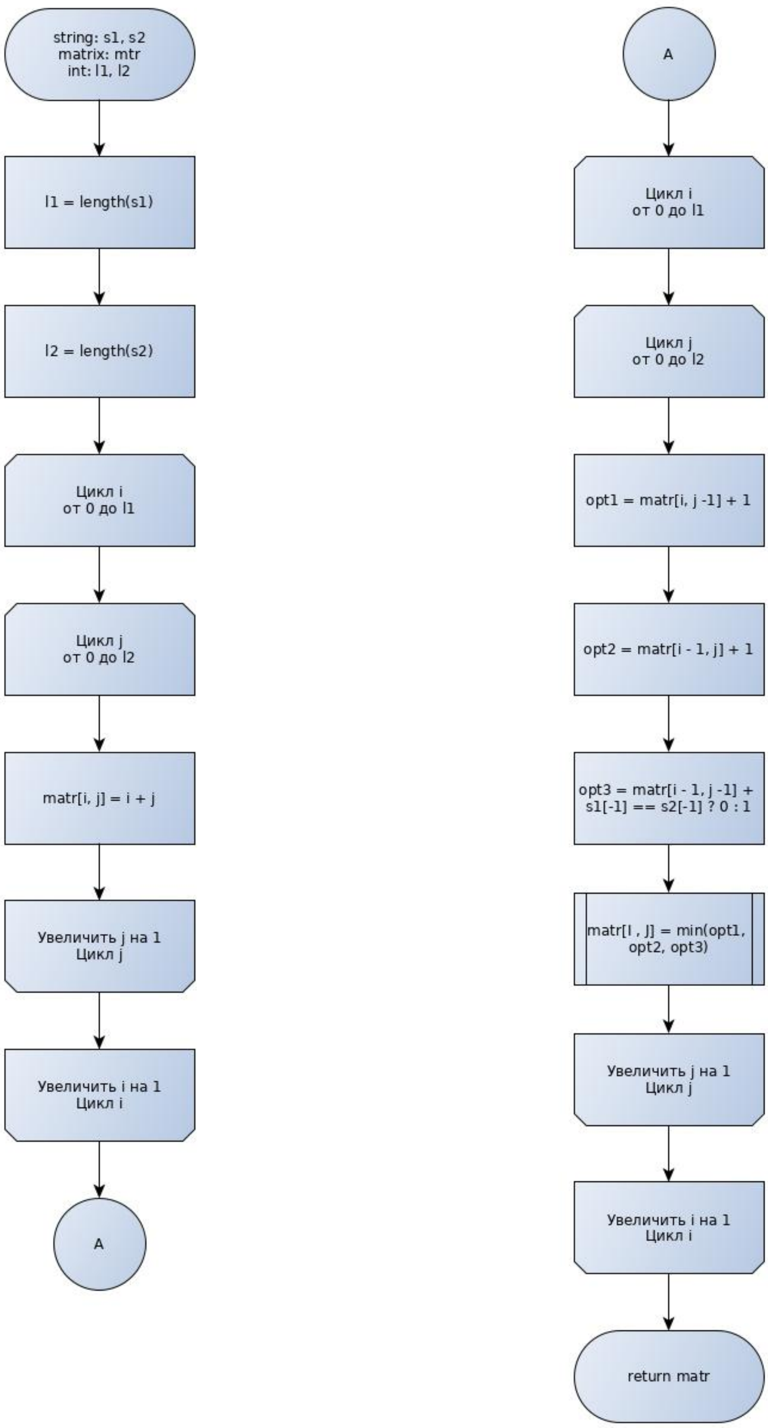


Рис. 1: Схема алгоритма нахождения расстояния Дамерау-Левенштейна(итеративного)

0.2.2 Вывод

В данном разделе были рассмотрены схемы алгоритмов нахождения расстояния Левенштейна, Дамерау-Левенштейна, а также рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна.

0.3 Технологическая часть

0.3.1 Требования к программному обеспечению

Входные данные: str1 - первое слово, str2 - второе слово.
Выходные данные: значение расстояния между двух слов.

0.3.2 Средства реализации

В качестве языка был выбран python, для вычисления памяти был использован метод sys.getsizeof(), возвращающий память, затрачиваемую на объект. Для замеров времени была использована функция tick(), приведенная ниже:

Листинг 1: Функция нахождения тиков

```
unsigned long long tick(void)
{
    unsigned long long d;
    __asm__ __volatile__ ("rdtsc" : "=A" (d) );
    return d;
}
```

Для подключения ее в модуль python была создана библиотека libtick.so и подключена с помощью модуля ctypes.

0.3.3 Листинг кода

Листинг 2: Функция нахождения расстояния Левенштейна итеративно

```
def Livenstein_matr(s1, s2):
  l1 = len(s1)
  l2 = len(s2)
  matr = [[i + j for i in range(l2 + 1)] for j in range(l1 + 1)]

  for i in range(1, l1 + 1):
      for j in range(1, l2 + 1):
          opt1 = matr[i][j - 1] + 1
          opt2 = matr[i - 1][j] + 1
          opt3 = matr[i - 1][j - 1] + (0 if s1[i - 1] == s2[j - 1]
              else 1)
          matr[i][j] = min(opt1, opt2, opt3 )

  res = matr[l1][l2]
  return res
```

Листинг 3: Функция нахождения расстояния Дамерау-Левенштейна итеративно

```
def Livenstein_Damerau_matr(s1, s2):
    l1 = len(s1)
    l2 = len(s2)
    matr = [[i + j for i in range(l2 + 1)] for j in range(l1 + 1)]

    for i in range(1, l1 + 1):
        for j in range(1, l2 + 1):
            opt4 = math.inf
            opt1 = matr[i][j - 1] + 1
            opt2 = matr[i - 1][j] + 1
            opt3 = matr[i - 1][j - 1] + (0 if s1[i - 1] == s2[j - 1] else 1)
            if i > 1 and j > 1 and s1[i - 2] == s2[j - 1] and s1[i - 1] == s2[j - 2]:
                opt4 = matr[i - 2][j - 2] + 1

            matr[i][j] = min(opt1, opt2, opt3, opt4)

    res = matr[l1][l2]
    return res
```

Листинг 4: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
def Livenstein_Damerau_recur(s1, s2):
    if s1 == "" and s2 == "":
        return 0
    elif s1 != "" and s2 == "":
        return len(s1)
    elif s1 == "" and s2 != "":
        return len(s2)

    l1 = len(s1)
    l2 = len(s2)

    opt4 = math.inf
    opt1 = Livenstein_Damorou_recur(s1, s2[:-1]) + 1
    opt2 = Livenstein_Damorou_recur(s1[:-1], s2) + 1
    opt3 = Livenstein_Damorou_recur(s1[:-1], s2[:-1]) + (0 if s1[l1 - 1] == s2[l2 - 1] else 1)
    if l1 > 1 and l2 > 1 and s1[l1 - 1] == s2[l2 - 2] and s1[l1 - 2] == s2[l2 - 1]:
        opt4 = Livenstein_Damorou_recur(s1[:-2], s2[:-2]) + 1

    res = min(opt1, opt2, opt3, opt4)
    return res
```

0.3.4 Тестирование

Было организовано тестирование с помощью заранее подготовленных данных в виде пары строк и расстояния по Левенштейну и Дамерау-Левенштейну. Вот эти данные:

s1	s2	р. Левенштейна	р. Дамерау-Левенштейна
‘ser‘	‘gey"	2	2
‘ser‘	‘sre"	2	1
‘ser‘	‘guy"	3	3
‘ser‘	‘esg"	3	2
‘sergeyser‘	‘sergey‘	3	3
“	‘sergey‘	6	6
“	“	0	0
“	‘a‘	1	1
‘a‘	‘b‘	1	1
‘serg‘	‘segr‘	2	1

Таблица 1. Тестовые данные для алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Все тесты дали положительный результат, что подтвердило правильность работы программы.

0.3.5 Сравнительный анализ потребляемой памяти

В итеративном алгоритме нахождения расстояния Левенштейна используются:

Структура данных	Занимаемая память (байты)
Матрица	40 + 8 * [len(str1) + 1] + [len(str1) + 1] * [40 + 8 * (len(str 2) + 1)]
2 строки	2 * [49 + len(str)]
5 вспомогательных переменных(int)	140
2 счетчика (int)	56

Таблица 2. Память, потребляемая структурами в алгоритме Левенштейна

В итеративном алгоритме нахождения расстояния Дамерау-Левенштейна используются:

Структура данных	Занимаемая память (байты)
Матрица	40 + 8 * [len(str1) + 1] + [len(str1) + 1] * [40 + 8 * (len(str 2) + 1)]
2 строки	2 * [49 + len(str)]
6 вспомогательных переменных(int)	166
2 счетчика (int)	56

Таблица 3. Память, потребляемая структурами в алгоритме Дамерау-Левенштейна

В рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна используются:

Структура данных	Занимаемая память (байты)
4 вспомогательных переменных (int)	112
2 строки	2 * [49 + len(str)]

Таблица 4. Память, потребляемая структурами в рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна

Максимальная глубина рекурсивного вызова функции - сумма длин двух слов.

0.3.6 Оценка потребляемой памяти на 4 и 1000 символах

Оценим алгоритмы на словах длинной 4 символа:

Структура данных	Занимаемая память (байты)
Матрица	480
2 строки	106
5 вспомогательных переменных(int)	140
2 счетчика (int)	56
Всего	**780**

Таблица 5. Память, потребляемая структурами в алгоритме нахождения расстояния Левенштейна

Структура данных	Занимаемая память (байты)
Матрица	480
2 строки	106
6 вспомогательных переменных(int)	168
2 счетчика (int)	56
Всего	**808**

Таблица 6. Память, потребляемая структурами в алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
4 вспомогательных переменных (int)	140 * 8(максимальная глубина вызовов) = 896
2 строки	106 * 4/2(Усредненное значение) = 212
Всего	**1108**

Таблица 7. Память, потребляемая структурами в рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна

Оценим алгоритмы на словах длинной 1000 символов:

Структура данных	Занимаемая память (байты)
Матрица	8 064 096
2 строки	106
5 вспомогательных переменных(int)	140
2 счетчика (int)	56
Всего	**8064398**

Таблица 8. Память, потребляемая структурами в алгоритме нахождения расстояния Левенштейна

Структура данных	Занимаемая память (байты)
Матрица	8 064 096
2 строки	106
6 вспомогательных переменных(int)	168
2 счетчика (int)	56
Всего	**8064426**

Таблица 9. Память, потребляемая структурами в алгоритме нахождения расстояния Дамерау-Левенштейна

Структура данных	Занимаемая память (байты)
4 вспомогательных переменных (int)	140 * 2000(максимальная глубина вызовов) = 280000
2 строки	2098 * 1000/2(Усредненное значение) = 1049000
Всего	**1329000**

Таблица 10. Память, потребляемая структурами в рекурсивном алгоритме нахождения расстояния Дамерау-Левенштейна

По таблицам мы видим, что и при длине слов 4, и при длине слов 1000 итеративные реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна сравнимы по потребляемой памяти, причем разница между ними всегда 28 байт. В свою очередь, рекурсивная реализация Дамерау-Левенштейна при 4 потребляет больше памяти, а при 1000 - меньше.

0.3.7 Вывод

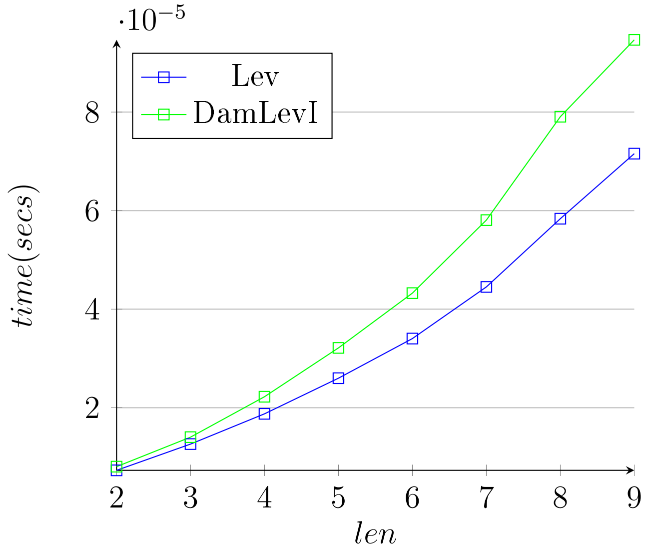
В технологической части была предоставлены реализации алгоритмов нахождения расстояния Левенштейна в итеративной форме, а также нахождения расстояния Дамерау-Левенштейна в итеративной и рекурсивной формах. Помимо этого, были предоставлены результаты тестов на правильность данных реализаций, сравнительный анализ потребляемой памяти всех реализаций на слов длиной 4 и 1000. Было выявлено, что и при длине слов 4, и при длине слов 1000 итеративные реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна сравнимы по потребляемой памяти, причем разница между ними всегда 28 байт(3 и 0.0001 процента при длине слова 4 и 1000 соответственно). Рекурсивная реализация же при длине слов 4 потребляет на 37 процентов больше памяти, в то время как при длине слов 1000 она потребляет меньше памяти на 84 процента.

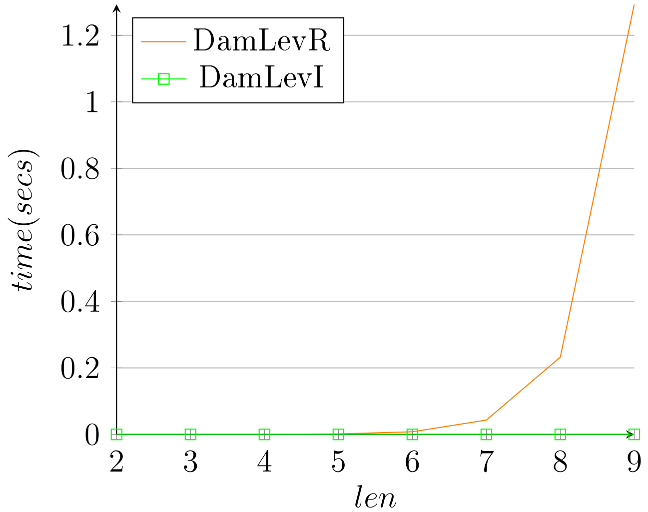
0.4 Экспериментальная часть

0.4.1 Постановка эксперимента

Должны быть произведены замеры времени работы каждого из алгоритмов при длинах строк от 2 до 9. Каждый тест должен быть был проведен 100 раз, как результат должно быть взято среднее значение для уменьшения роли случайных факторов в итоге.

0.4.2 Результаты эксперимента





Таблицы с данными для графиков

Длина	Левенштейна	м. Дамерау-Левенштейна	р. Дамерау-Левенштейна
2	7.2e-06	8.0e-06	1.0e-05
3	1.2e-05	1.4e-05	5.7e-05
4	1.8e-05	2.2e-05	2.8e-04
5	2.5e-05	3.2e-05	1.4e-03
6	3.4e-05	4.3e-05	7.1e-03
7	4.4e-05	5.8e-05	4.3e-02
8	5.8e-05	7.9e-05	0.2
9	7.1e-05	9.4e-05	1.3

Таблица 11. Время, затрачиваемое различными алгоритмами на обработку строк длин от 2 до 9(в секундах).

0.4.3 Вывод

По первому графику видно, что временные затраты на итеративный алгоритмы Левенштейна и Дамерау-Левенштейна сравнимы, но при этом алгоритм Дамерау-Левенштейна всегда медленнее. Из второго графика мы замечаем то, что рекурсивный алгоритм Дамерау-Левенштейна на порядки более затратный по времени, чем итеративный, начиная с длины строки в 5.

Заключение

В ходе данной лабороторной работы мною были реализовани алгоритмы Левенштейна в матричной форме и Дамерау-Левенштейна в матричной и рекурсивной форме. В ходе проверки на временные затраты было выявлено, что матричные реализации алгоритма Левенштейна и Дамерау-Левенштейна сравнимы по затрачиваемым процессорным ресурсам при длинах слов от 2 до 9(разница между ними растет от 11 до 25 процентов в пользу Левенштейна). По используемой памяти разница меньше: от 0 до 3 процентов. Также было выявлено, что начиная со строк длиной в 3 символа рекурсивный вариант Дамерау-Левенштейна на порядки более затратный по процессорному времени, чем матричный(от 12.6 до 1.3x10^4 раз). Это вызвано тем, что в рекурсивном виде алгоритм одни и те же расчеты производит по несколько раз, так как расстояние между оними и теми же промежуточными словами может быть запрошено в нескольких независимо вызванных функциях, в то время как в матричном варианте все промежуточные расчеты записываются в матрицу и не пересчитываются. Говоря о памяти: рекурсивная форма нахождения расстояния Дамерау-Левенштейна при длине строки в 1000 на 84 процента меньше памяти, хотя при длине строки 4 потребляет на 34 процента больше памяти.

Список литературы

Дж. Макконнелл. Анализ алгоритмов. Активный обучающий подход.-М.:Техносфера, 2009.
Нечёткий поиск в тексте и словаре // [Электронный ресурс]. Режим доступа: https://habr.com/ru/post/114997/ (дата обращения: 2.10.19).
Нечеткий поиск, расстояние левенштейна алгоритм // [Электронный ресурс]. Режим доступа: https://steptosleep.ru/antananarivo-106/ (дата обращения: 2.10.19).

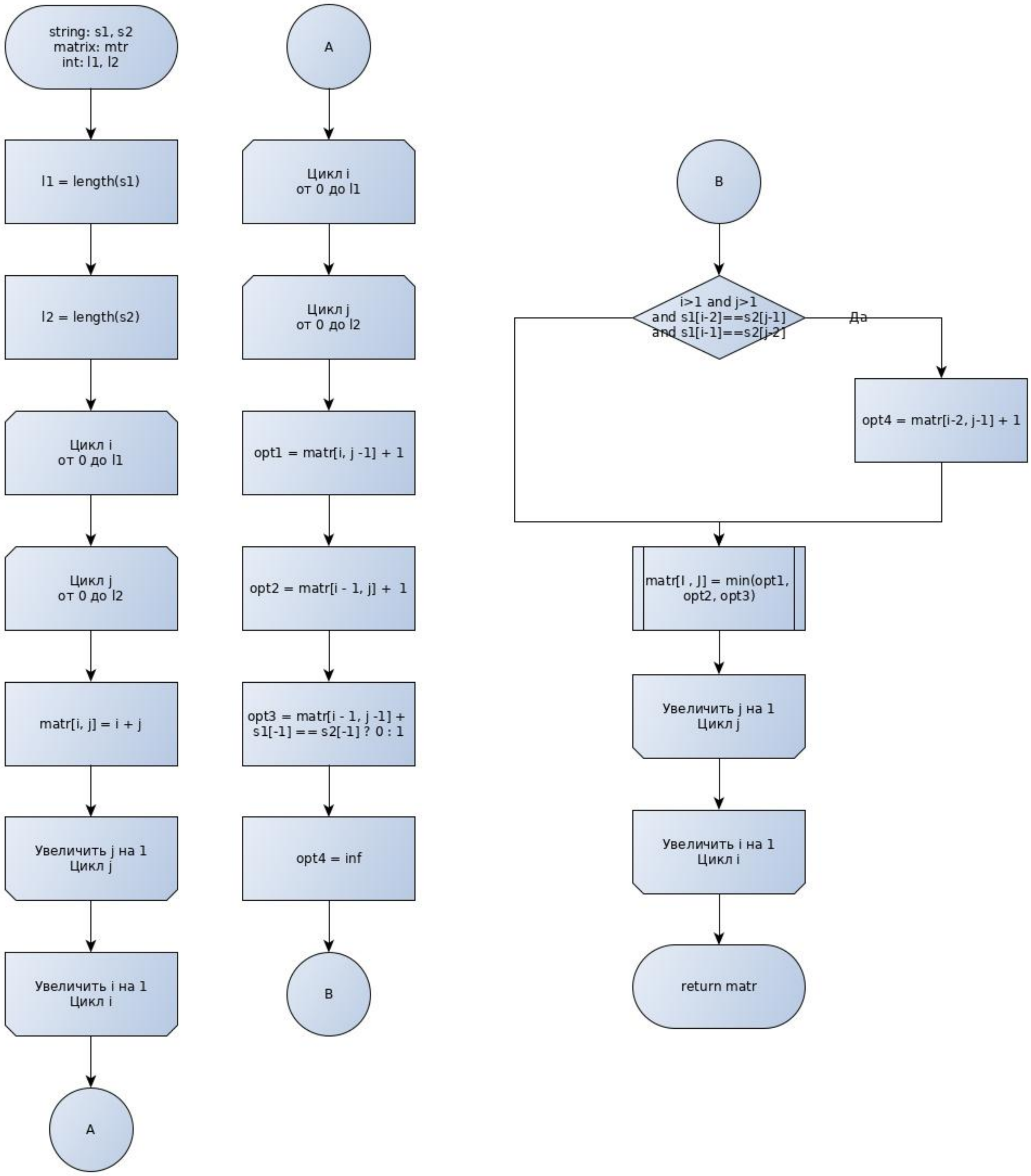


Рис. 2: Схема алгоритма нахождения расстояния Дамерау-Левенштейна(итеративного)

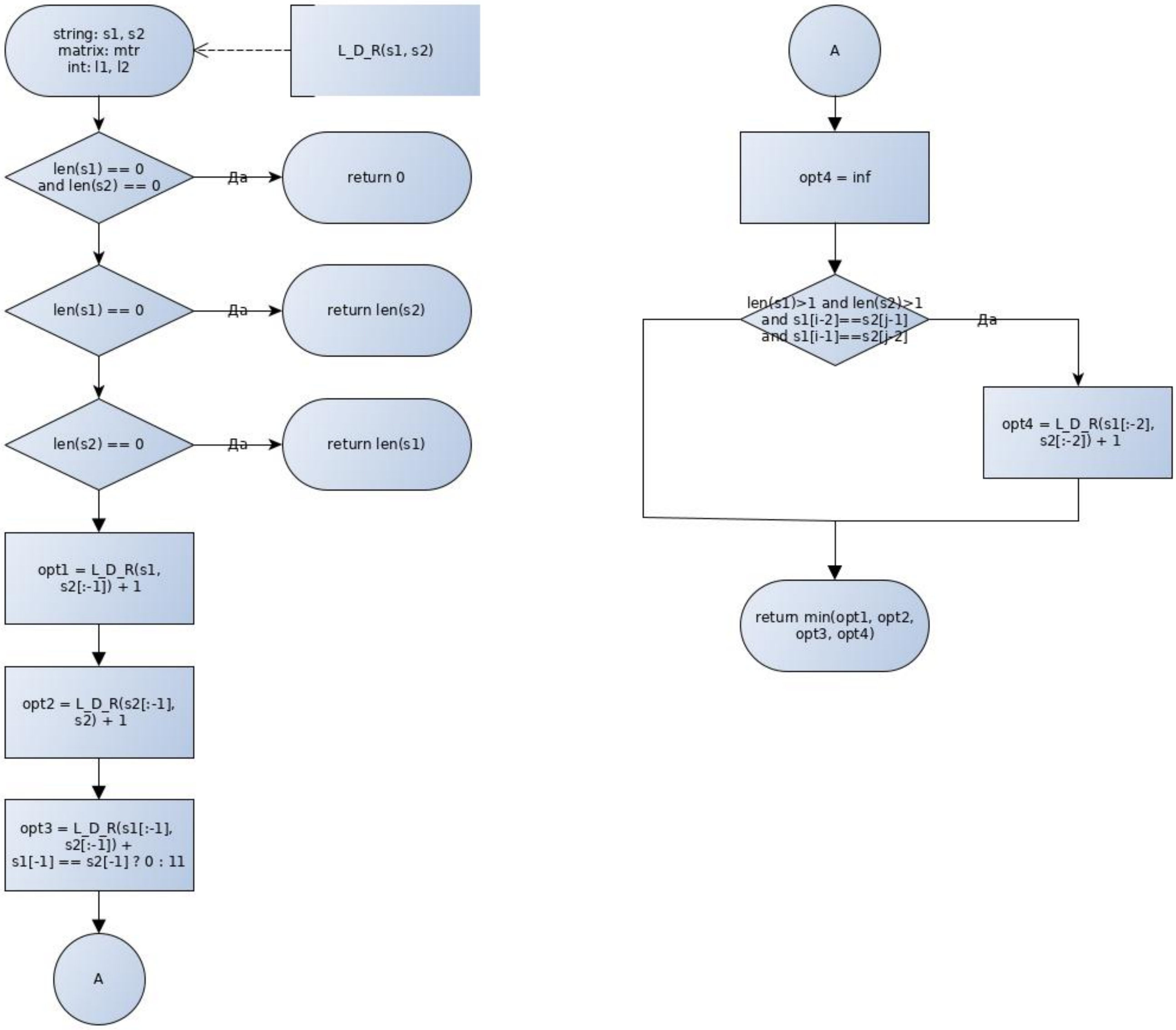


Рис. 3: Схема алгоритма нахождения расстояния Дамерау-Левенштейна(рекурсивного)

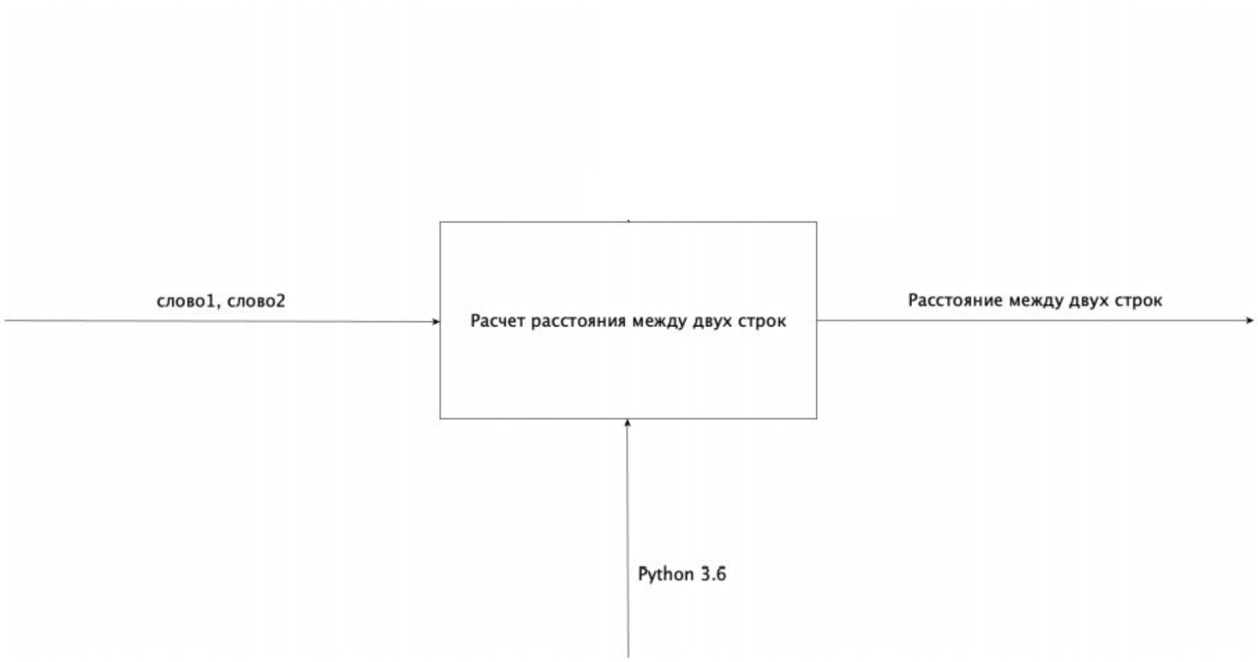


Рис. 4: IDEF0-диаграмма, описывающая алгоритм нахождения расстояния Левенштейна