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 Челядинов И.Д.

Группа ИУ7-53Б

Оценка (баллы) _____

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2020 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 применяется в теории информации и компьютерной лингвистике в следующих случаях:

- исправления ошибок в слове(поисковая строка браузера);
- сравнения текстовых файлов утилитой diff;
- в биоинформатике для сравнения генов, хромосом и белков.

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачами данной лабораторной являются:

- изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
- применение метода динамического программирования для матричной реализации указанных алгоритмов;
- получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и двух алгоритмов в рекурсивной версии;
- сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
- 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;
- описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1. Аналитическая часть

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

ействия обозначаются слдующим образом:

1. D (англ. delete) — удалить;
2. I (англ. insert) — вставить;
3. R (replace) — заменить;
4. M(match) - совпадение.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле (1.1):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(i,j-1) + 1, & \\ D(i-1,j) + 1, & j > 0, i > 0 \\ D(i-1,j-1) + m(S_1[i], S_2[j]) & \\ ), & \end{cases} \tag{1.1}$$

где $m(a,b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{\, a, b, c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле (1.2):

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \\ D(i-2,j-2)+m(S_1[i],S_2[i]), \end{cases} & \begin{array}{l} \text{, если } i,j>0 \\ \text{и } S_1[i] = S_2[j-1] \\ \text{и } S_1[i-1] = S_2[j] \end{array} \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \end{cases} & \text{, иначе} \end{cases} \tag{1.2}$$

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

2. Конструкторская часть

Требования к программе:

1. На вход подаются две строки.
2. Буквы в верхнем регистре и в нижнем регистре считаются разными.
3. Две пустые строки - корректный ввод, программа не должна аварийно завершаться.
4. Для всех алгоритмов выводиться процессорное время исполнения.
5. Для всех алгоритмов кроме Левенштейна с рекурсивной реализацией должна выводиться матрица.

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы следующих алгоритмов Левенштейна : рекурсивного алгоритма Левенштейна (рис. 2.1), рекурсивного алгоритма Левенштейна с матричной оптимизацией (рис. 2.2), матричного алгоритма Левенштейна (рис. 2.3), матричного алгоритма Левенштейна (рис. 2.4).[1]

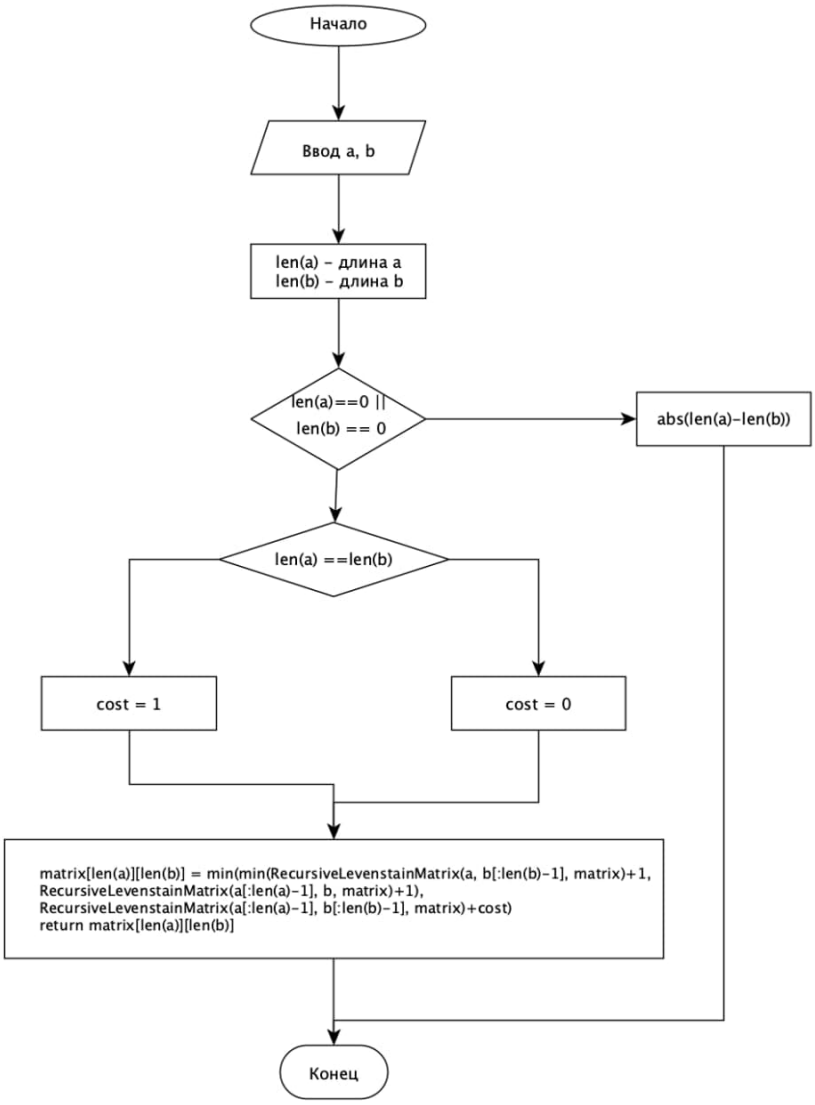


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

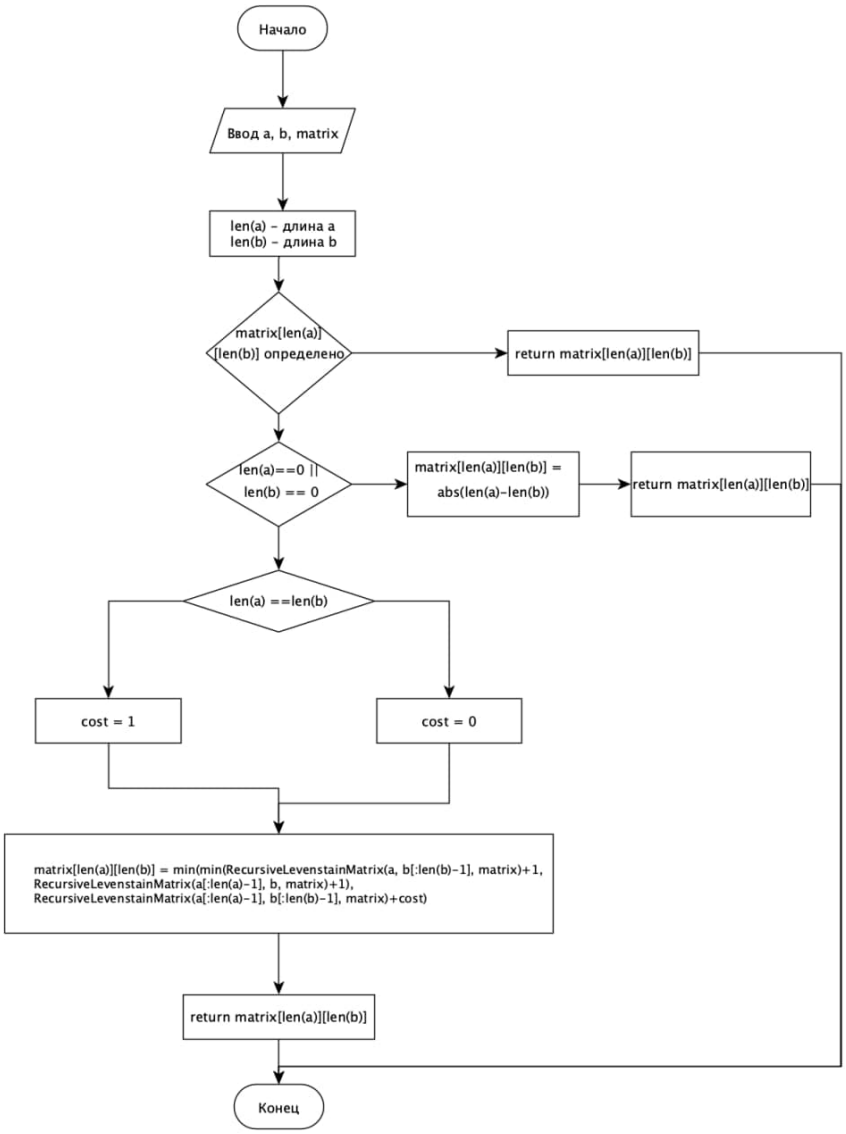
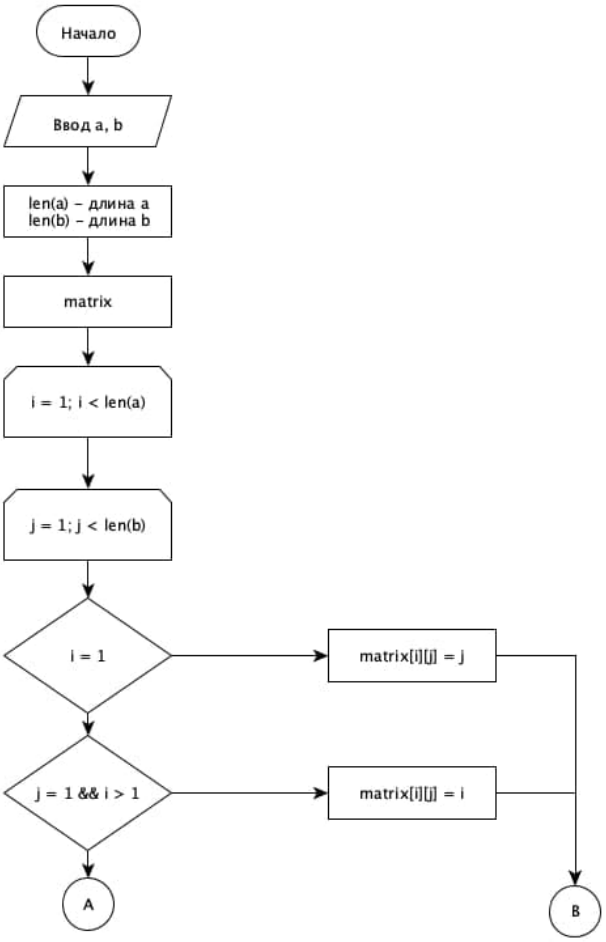


Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с матричной оптимизацией


Начало
Ввод a, b
len(a) – длина a
len(b) – длина b
matrix
i = 1; i < len(a)
j = 1; j < len(b)
i = 1
matrix[i][j] = j
j = 1 && i > 1
matrix[i][j] = i
A
B

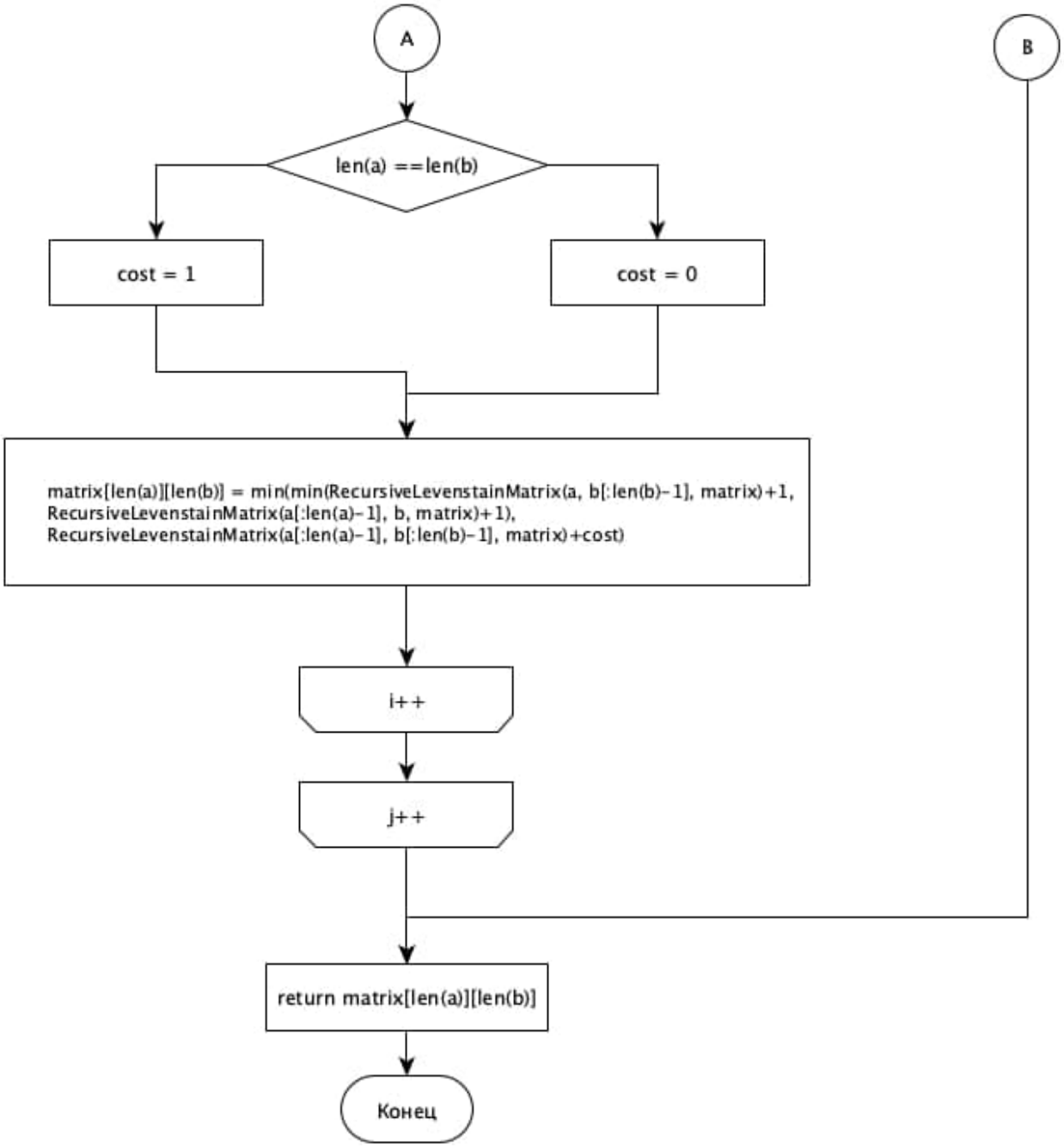
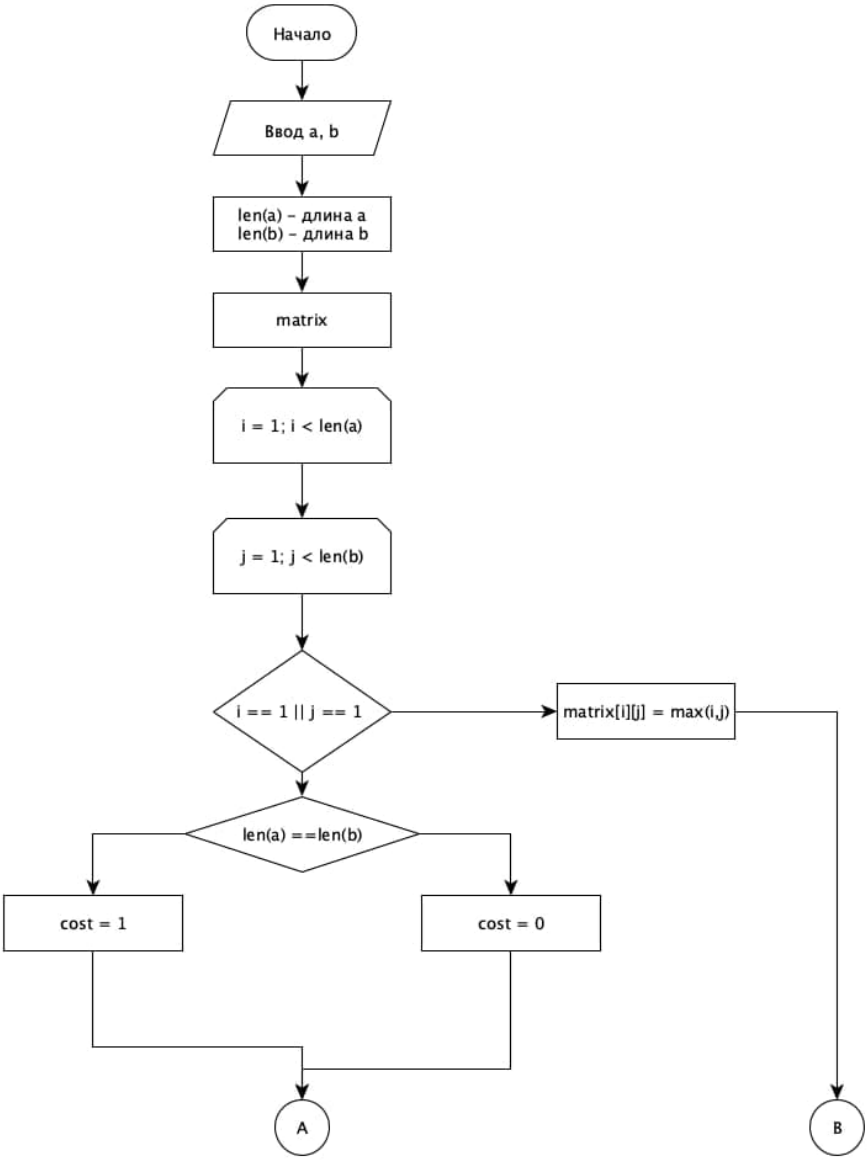


Рис. 2.3: Схема матричного алгоритма нахождения расстояния Левенштейна


Начало
Ввод a, b
len(a) – длина a
len(b) – длина b
matrix
i = 1; i < len(a)
j = 1; j < len(b)
i == 1 || j == 1
matrix[i][j] = max(i,j)
len(a) ==len(b)
cost = 1
cost = 0
A
B

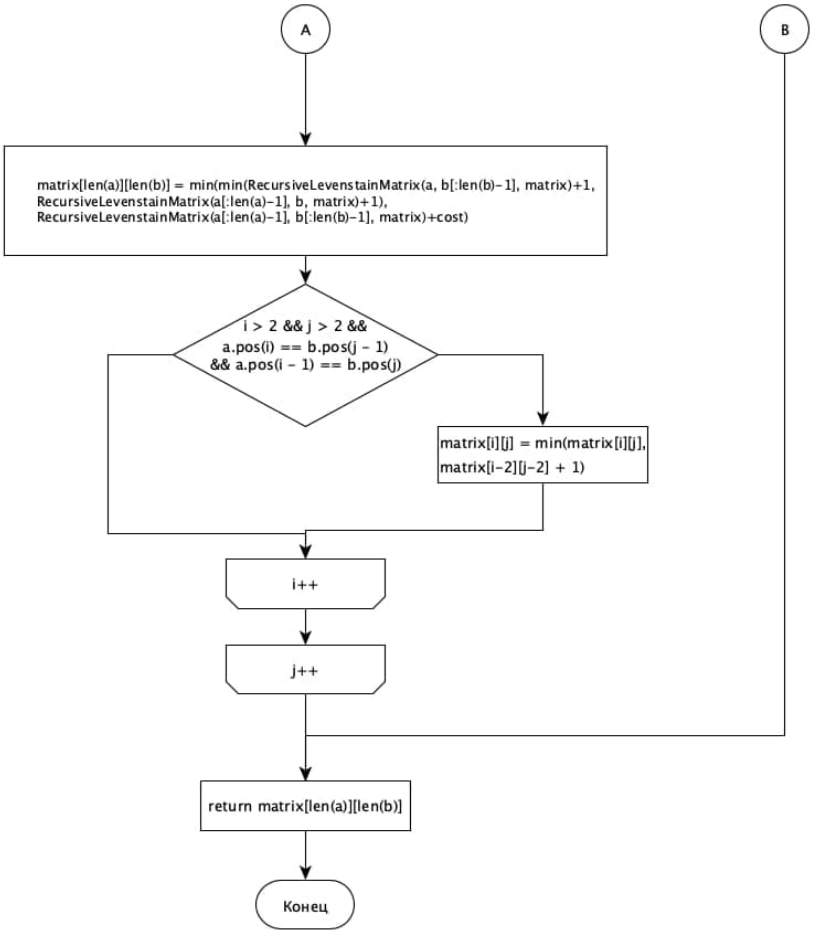


Рис. 2.4: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

Вывод

В данном разделе были рассмотрены требования к работе, а также схемы алгоритмов нахождения редакционного расстояния с помощью алгоритмов Левенштейна и Дамерау-Левенштейна.

3. Технологическая часть

3.1 Выбор ЯП

Для реализации программы был выбран язык программирования Golang в связи с потребностью практики разработки на нем. Среда разработки - Goland .[2] [3]

3.2 Реализация алгоритма

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
func RecursiveLevenstain(a, b []rune) int {
  var cost int
  if len(a) == 0 || len(b) == 0 {
    return int(math.Abs(float64(len(a) - len(b))))
  }
  cost = 1
  if a[len(a)-1] == b[len(b)-1] {
    cost = 0
  }

  return min(min(RecursiveLevenstain(a, b[:len(b)-1])+1,
    RecursiveLevenstain(a[:len(a)-1], b)+1), RecursiveLevenstain(a[:len(a)
       -1], b[:len(b)-1])+cost)
}
```

Листинг 3.2: Функция нахождения расстояния Левенштейна рекурсивно с матрицей

```
func RecursiveLevenstainMatrix(a, b []rune, matrix [][]int) int {
  var cost int
  if len(a) == 0 || len(b) == 0 {
    matrix[len(a)][len(b)] = int(math.Abs(float64(len(a) - len(b))))
  }
  if matrix[len(a)][len(b)] != -1 {
```

```
7       return matrix[len(a)][len(b)]
8     }
9     cost = 1
10    if a[len(a)-1] == b[len(b)-1] {
11      cost = 0
12    }
13
14    matrix[len(a)][len(b)] = min(min(RecursiveLevenstainMatrix(a, b[:len(b)
          -1], matrix)+1,
15      RecursiveLevenstainMatrix(a[:len(a)-1], b, matrix)+1),
16      RecursiveLevenstainMatrix(a[:len(a)-1], b[:len(b)-1], matrix)+cost)
17    return matrix[len(a)][len(b)]
18  }
```

Листинг 3.3: Функция нахождения расстояния Левенштейна матрично

```
1  func MatrixLevenstain(a, b []rune, matrix [][]int) int {
2    var cost int
3    for i := 0; i < len(a)+1; i++ {
4      for j := 0; j < len(b)+1; j++ {
5        if i == 0 {
6          matrix[i][j] = j
7        } else if j == 0 && i > 0 {
8          matrix[i][j] = i
9        } else {
10         cost = 1
11         if a[i-1] == b[j-1] {
12           cost = 0
13         }
14         matrix[i][j] = min(min(matrix[i][j-1]+1,
15           matrix[i-1][j]+1),
16           matrix[i-1][j-1]+cost)
17       }
18     }
```

```
19    }
20    return matrix[len(a)][len(b)]
21  }
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
1  func DamerauLevenstainMatrix(a, b []rune, matrix [][]int) int {
2    var cost int
3    for i := 0; i < len(a)+1; i++ {
4      for j := 0; j < len(b)+1; j++ {
5        if i == 0 || j == 0 {
6          matrix[i][j] = max(i, j)
7        } else {
8          cost = 1
9          if a[i-1] == b[j-1] {
10           cost = 0
11         }
12         matrix[i][j] = min(min(matrix[i][j-1]+1,
13           matrix[i-1][j]+1),
14           matrix[i-1][j-1]+cost)
15         if j > 1 && i > 1 &&
16           a[i-1] == b[j-2] &&
17           a[i-2] == b[j-1] {
18           matrix[i][j] = min(min(matrix[i][j], matrix[i-2][j-2]+1),matrix[i
                 ][j])
19         }
20       }
21     }
22   }
23   return matrix[len(a)][len(b)]
24 }
```

Листинг 3.5: Функции поиска минимума и максимума

```
1  func min(a, b int)int {
```

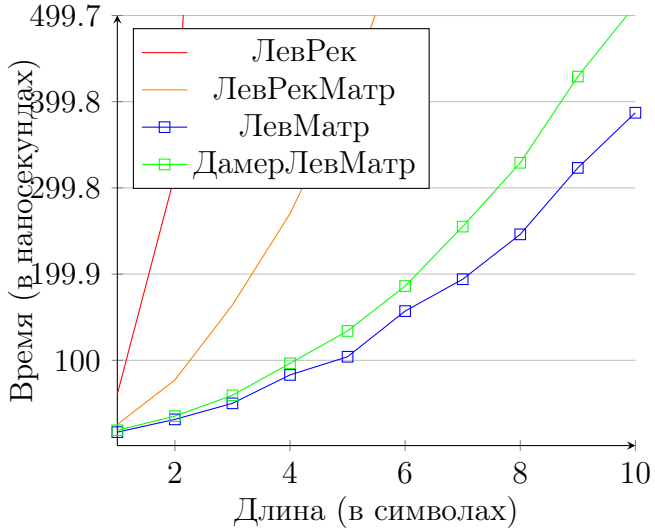
```
2   if a > b{
3     return b
4   }
5   return a
6 }
7 func max(a, b int) int{
8   if a < b{
9     return b
10  }
11  return a
12 }
```

4. Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов. Замер производился на ноутбуке Macbook pro 13 на базе процессоре Intel core i5, который обладает 1.4 GHz тактовой частоты, а также 8 гигабайтами оперативной памяти.[4][5] Для произведения замера времени использовалась система тестирования Benchmark языка Golang. Данная система тестирования позволяет измерить скорость работы алгоритма с минимальной погрешностью.

Рис. 4: зависимость времени исполнения от длины слова



Наиболее быстрым является алгоритм Левенштейна с матрицей; в нем требуется только (m + 1)*(n + 1) операций заполнения ячейки матрицы. Рекурсивная версия алгоритма крайне сильно проигрывает матричному алгоритму уже на строках длиной 10 символов, из-за большой глубины рекурсии и многочисленных повторов. Рекурсивная версия с матрицей не теряет в своей производительности настолько сильно, поскольку в ней исключаются повторы благодаря кэшированию результатов вычисления ячеек матрицы. Заметим, что алгоритм Дамерау-Левенштейна выполняется относительного немного дольше алгоритма Левенштейна, т.к. в нем добавлена дополнительная проверка (проверка на транспозицию символов). В среднем алгоритм Дамерау-Левенштейна с матрицей работает на 10% дольше, чем матричный алгоритм Левенштейна.

Таблица 4.1: время исполнения в наносекундах

len	Lev(R)	Lev(RM)	Lev(M)	DamLev(M)
1	12	25.7	16.7	18.7
2	60.7	77.0	31.4	35.3
3	315	164	50.1	59.4
4	1584	270	83	96.5
5	8347	409	104	134
6	46547	599	157	186
7	243649	781	194	255
8	1363796	1045	246	329
9	7242114	1289	323	429
10	40095403	1595	387	512

4.2 Тесты

Проведем тестирование программы. В столбцах "Ожидание"и "Результат"4 числа соответствуют рекурсивному алгоритму нахождения расстояния Левенштейна, рекурсивно-матричному алгоритму нахождения расстояния Левенштейна, матричному алгоритму расстояния Левенштейна, матричному алгоритму нахождения расстояния Дамерау-Левенштейна.

Таблица 4.2: Таблица тестовых данных

№	Слово №1	Слово №2	Ожидание	Результат
1			0 0 0 0	0 0 0 0
2	1234	2143	3 3 3 2	3 3 3 2
3	dtt	pkk	3 3 3 3	3 3 3 3
4	cat	dog	3 3 3 3	3 3 3 3
5		olololol	8 8 8 8	8 8 8 8
6	lolololo	0	8 8 8 8	8 8 8 8
7	asdsas	asdssa	2 2 2 1	2 2 2 1

Вывод

Закодировав исследованные алгоритмы, была получена итоговая программа. Протестированная программа выдала результаты, соответсвующие теоретическим данным.

Заключение

В ходе лабораторной работы были изучены алгоритмы Левенштейна и Дамерау-Левенштейна - алгоритмы нахождения редакционного расстояния между строками, получены практические навыки реализации указанных алгоритмов в матричной и рекурсивных версиях. Также был изучен метод динамического программирования на материале данных алгоритмов.

Экспериментально было подтверждено различие по временным затратам рекурсивной и линейной реализаций алгоритма Левенштейна при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на строках различной длины.

Самым быстродейственным является алгоритм Левенштейна с матрицей. Рекурсивная версия алгоритма крайне сильно проигрывает матричному алгоритму уже на строках длиной 10 символов, из-за большой глубины рекурсии и многочисленных повторов. Рекурсивная версия с матрицей не теряет в своей производительности настолько сильно, поскольку в ней исключаются повторы благодаря кэшированию результатов вычисления ячеек матрицы. Линейный алгоритм Дамерау-Левенштейна выполняется относительного незначительно дольше линейного алгоритма Левенштейна, поскольку в нем добавлена дополнительная проверка (проверка на транспозицию символов): в среднем матричный алгоритм Левенштейна работает на 10% быстрее, чем алгоритм Дамерау-Левенштейна с матрицей.

Подытожив, матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах и задачах.

Литература

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] Саммерфильд Марк. Программирование на Go. Разработка приложений XXI века. –М.: ДМК Пресс, 2013, 2013. Т. 580. С. 130–216.

[3] Среда разработки Goland. Режим доступа: https://www.jetbrains.com/go/promo/ (дата обращения: 29.9.2020).

[4] Технические характеристики ноутбука Apple MacBook Pro. Режим доступа: https://www.apple.com/macbook-pro-13/ (дата обращения: 5.10.2020).

[5] Процессор Intel® Core™ i5 10 gen. [Электронный ресурс]. Режим доступа: https://https://www.intel.ru/content/www/ru/ru/ products/docs/processors/core/ 10th-gen-processors.html (дата обращения: 30.09.2020).

[6] Absolute time. Режим доступа: https://https://developer.apple.com/ documentation/kernel/1462446-machabsolutetime (дата обращения: 20.09.2020).