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 Расстояние Левенштейна

Студент Романов А.В.

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 для:

- исправления ошибок в слове
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов, хромосом и белков

Целью данной лабораторной работы:

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

Задачи данной лабораторной работы:

1. Изучение алгоритмов Левенштейна и Дамерау-Левенштейна;
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 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;

6. 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).

Обозначение операций

1. D (delete) – удалить,
2. I (insert) – вставить,
3. R (replace) – заменить,
4. M (match) – совпадение символов.

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:

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

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:

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

1.1 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 вводу:

1. На вход подаются две строки

Требования к программе:

1. ПО должно выводить полученное расстояние и вспомогательные матрицы;
2. ПО должно выводить потраченную память и время;

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

Рис. 2.2: Схема матричного алгоритма нахождения расстояния Левенштейна

Рис. 2.3: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

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

3 | Технологическая часть

3.1 Выбор ЯП

Для реализации программы нахождения расстояние Левенштейна я выбрал язык программирования Haskell. Так как Haskell использует ленивые вычисления, рекурсивные алгоритмы крайне неэффективы: на строках длинной более 10 уже полностью заполняется оперативная память.

3.2 Реализация алгоритма

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
1  levenshteinRecursion :: String -> String -> (Distance,
       Depth)
2  levenshteinRecursion s1 s2 = _recursion s1 s2 0
3    where _recursion s1 "" n = (length s1, n)
4          _recursion "" s2 n = (length s2, n)
5          _recursion s1 s2 n = (score, depth) where
6          (insert, curr1) = _recursion (init s1) s2 (n + 1)
7          (delete, curr2) = _recursion s1 (init s2) (n + 1)
8          (replace, curr3) = _recursion (init s1) (init s2) (n +
             1)
9
10         match = if last s1 == last s2 then 0 else 1
11         score = min3 (insert + 1) (delete + 1) (replace + match
             )
12         depth = max3 curr1 curr2 curr3
```

Листинг 3.2: Функция нахождения расстояние Левенштейна рекурсивно с мемоизацией

```
levenshteinMemoized :: String -> String -> (Distance, Depth)
levenshteinMemoized s1 s2 = _memoized s1 s2 matrix 0
  where matrix = fromList (length s1 + 1) (length s2 + 1) $ repeat (-1)
    _memoized s1 "" _ n = (length s1, n)
    _memoized "" s2 _ n = (length s2, n)
    _memoized s1 s2 mtr n = (score, depth) where
    score = min3 (insert + 1) (delete + 1) (replace + match)
    memoized = (getElem (length s1) (length s2) mtr, n + 1)
    new_mtr = setElem score (length s1, length s2) mtr

    (insert, curr1) = if fst memoized == -1 then _memoized (init s1) s2 new_mtr (n + 1)
      else memoized
    (delete, curr2) = if fst memoized == -1 then _memoized s1 (init s2) new_mtr (n + 1)
      else memoized
    (replace, curr3) = if fst memoized == -1 then _memoized (init s1) (init s2) new_mtr (n + 1)
      else memoized

    match = if last s1 == last s2 then 0 else 1
    depth = max3 curr1 curr2 curr3
```

Листинг 3.3: Функция нахождения расстояния Левенштейна итеративно

```
levenshteinIterative :: String -> String -> Matrix Int
levenshteinIterative s1 s2 = fromLists $ reverse $ foldl
  (\mtr i -> if head mtr == [] then [[0..length s1]] else calcRow mtr i : mtr) [[]] [0..length s2]
  where calcRow mtr i = foldl (\row j ->
      row ++ if length row == 0 then [length mtr] else [
        min3 (last row + 1) (head mtr !! j + 1) $ head mtr !! (j - 1) +
          if s1 !! (j - 1) == s2 !! (i - 1) then 0 else 1]
      ) [] [0..length s1]
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
1  damerauLevenshtein :: String -> String -> Matrix Int
2  damerauLevenshtein s1 s2 = fromLists $ reverse $ foldl
3    (\mtr i -> if head mtr == [] then [[0..length s1]] else
       calcRow mtr i : mtr) [[]] [0..length s2]
4    where cell mtr row i j = min3 (last row + 1) (head mtr !!
         j + 1) $ head mtr !! (j - 1) +
5        if s1 !! (j - 1) == s2 !! (i - 1) then 0 else 1
6        transposition i j = i > 1 && j > 1 && s1 !! (j - 1)
           == s2 !! (i - 2) && s1 !! (j - 2) == s2 !! (i - 1)
7        calcRow mtr i = foldl (\row j -> row ++ [
8          if length row == 0 then length mtr
9          else if transposition i j then min (cell mtr row i
             j) (((head $ tail mtr) !! i - 2) + 1)
10         else cell mtr row i j]
11       ) [] [0..length s1]
```

4 | Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Замеры времени работы каждого из алгоритмов.

Таблица 4.1: Время работы алгоритмов (в наносек)

len	Lev(R)	Lev(MR)	Lev(I)	DamLev(I)
10	5928	3724258	7456	4367560
20	16865	7224736	21854	8286833
30	62333	12123365	105445	12852145
50	372661	16940041	407763	18585284
100	1909255	23402008	1966658	24103230
200	9065189	32328258	11002094	27935583

вывать..

4.2 Тестовые данные

Таблица 4.2: Таблица тестовых данных

№	Первое слово	Второе слово	Ожидаемый результат	Полученный результат
1			0 0 0 0	0 0 0 0
2	kot	skat	2 2 2 2	2 2 2 2
3	kate	ktae	2 2 1 1	2 2 1 1
4	abacaba	aabcaab	4 4 2 2	4 4 2 2
5	sobaka	sboku	3 3 3 3	3 3 3 3
6	qwerty	queue	4 4 4 4	4 4 4 4
7	apple	aplpe	2 2 1 1	2 2 1 1
8		cat	3 3 3 3	3 3 3 3
9	parallels		9 9 9 9	9 9 9 9
10	bmstu	utsmb	4 4 4 4	4 4 4 4

Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

В результате исследований я пришла к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.