Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

Московский государственный технический университет имени Н.Э.Баумана
(МГТУ им. Н.Э.Баумана)

ОТЧЕТ
По лабораторной работе № 1
По курсу «Анализ алгоритмов»
на тему «Алгоритм Левенштейна»

Выполнил Студент: Московец Н.С
Группа: ИУ7-51

Москва, 2017

Оглавление

Постановка задачи

Изучить и реализовать алгоритм Левенштейна поиска редакционного расстояния между двумя строками с помощью матриц, а также рекурсивным методом, а также модифицировать его при условии добавлении операции перестановки элементов, сравнить реализованные алгоритмы.

Описание алгоритма

Расстояние Левенштейна (также редакционное расстояние или дистанция редактирования) между двумя строками — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Допускаются редакторские операции:

1. Замена символа
2. Вставка символа
3. Удаление символа
4. Совпадение символа

В различных реализациях алгоритма, каждая из операций может иметь свою стоимость. В дальнейшем, будем считать, что стоимость операций 1-3 равна 1, а операции 4 — 0.

Например, для строк “увлечение” и “развлечение” редакционное расстояние равно 4, что соответствует двум операциям вставки и двум операциям замены символа.

Расстояние Левенштейна и его обобщения активно применяется:

• для исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи).

• для сравнения текстовых файлов утилитой diff и ей подобными. Здесь роль «символов» играют строки, а роль «строк» — файлы.

• в биоинформатике для сравнения генов, хромосом и белков.

Редакционное расстояние для двух строк A и B можно вычислить рекуррентно по формуле:

$$D(A_n, B_m) = min[D(A_{n-1}, B_m) + 1,\ D(A_n, B_{m-1}) + 1,\ D(A_{n-1}, B_{m-1}) + \\ + \{\ 0,\ \text{если } A[m] = B[n];\ 1,\ \text{иначе}\ \}$$

Пример работы алгоритма:

	-	м	а	т	р	и	ц	а
-	0	1	2	3	4	5	6	7
м	1	0	1	2	3	4	5	6
е	2	1	1	2	3	4	5	6
т	3	2	2	1	2	3	4	5
р	4	3	3	2	1	2	3	4
а	5	4	3	3	2	2	3	3

Модифицированный алгоритм

Для вычисления редакционного расстояния иногда вводится дополнительная операция: перестановка (транспозиция) символов.

Существует несколько вариантов реализации работы алгоритма, поддерживающего данную операцию.

Простой способ учитывает только символы, которые находятся рядом во входных строках. Таким образом, модификация осуществляется только с

помощью ввода дополнительной проверки - если рядом расположенные символы в двух строках совпадают, алгоритм прибавляет к результату D[i-2][j-2] стоимость операции транспозиции. Однако, такой алгоритм будет некорректно обрабатывать строки, например, *abc* и *ca*, так как символы *a* и *c* расположены не рядом.

Корректная реализация алгоритма предусматривает запоминание последней рассмотренной позиции для каждого символа в строках. Так как построение осуществляется в матрице по строкам (в реализации, представленной в данной работе), то такое хранение поддерживается с помощью массива lastPosition, который хранит последнее вхождение каждого символа в первой строке, и индекса last, который запоминает последний индекс во второй строке для текущего рассматриваемого элемента из первой строки.

При этом, стоимость операции транспозициями в такой реализации определяется суммой расстояний между найденной парой символов в обеих строках и стоимости самой операции, т.е. 1.

Реализация

1. Базовый алгоритм

```
//Базовый алгоритм Левенштейна
int FindDistanceBase(const string &a, const string &b)
{
  int n, m;
  n = (unsigned int) a.size();
  m = (unsigned int) b.size();

  int D[n + 1][m + 1];

  //Инициализация матрицы
  D[0][0] = 0;
  for(int i = 0; i <= n; i++) {
     D[i][0] = i;
  }
  for(int j = 0; j <= m; j++) {
     D[0][j] = j;
  }

  for(int i = 1; i <= n; i++) {
     for(int j = 1; j <= m; j++)   {
        //вычисление редакционного расстояний для строк a[0, i - 1], b[0, j - 1]
        D[i][j] = min(D[i - 1][j] + 1, D[i][j - 1] + 1);
        D[i][j] = min(D[i - 1][j - 1] + (int)(a[i - 1] != b[j - 1]), D[i][j]);
     }
  }
  return D[n][m];
}
```

2. Модифицированный алгоритм (простой)

```
//Модифицированный алгоритм Левенштейна, учитывающий операцию транспозиции
//работает для символов, расположенных рядом друг с другом
int FindDistanceMod(const string &a, const string &b)
{
  int n, m;
  n = (unsigned int) a.size();
  m = (unsigned int) b.size();

  int D[n + 1][m + 1];

  //Инициализация матрицы
  D[0][0] = 0;
  for(int i = 0; i <= n; i++) {
     D[i][0] = i;
  }
  for(int j = 0; j <= m; j++) {
     D[0][j] = j;
  }

  for(int i = 1; i <= n; i++) {
     for(int j = 1; j <= m; j++)   {
        D[i][j] = min(D[i - 1][j] + 1, D[i][j - 1] + 1);
        D[i][j] = min(D[i - 1][j - 1] + (int) (a[i - 1] != b[j - 1]), D[i][j]);
        //проверка возможности транспозиции последних двух символов
        if(i > 1 && j > 1 && a[i - 2] == b[j - 1] && a[i - 1] == b[j - 2]) {
           D[i][j] = min(D[i - 2][j - 2] + 1, D[i][j]);
        }
     }
  }
  return D[n][m];
}
```

3. Модифицированный алгоритм (корректный)

```
//Модифицированный алгоритм Левенштейна, учитывающий операцию транспозиции
//корректно обрабатывает строки с операцией транспозиции для символов, не расположенных рядом
#define SYMBOL_COUNT 256 //количество всех возможных символов
int FindDistanceCoolMod(const string &a, const string &b)
{
  int n, m;
  n = (unsigned int) a.size();
  m = (unsigned int) b.size();

  int D[n + 2][m + 2];
  int INF = n + m;
  //Инициализация матрицы
  D[0][0] = INF;
```

```
  for(int i = 0; i <= n; i++) {
     D[i + 1][0] = INF;
     D[i + 1][1] = i;
  }
  for(int j = 0; j <= m; j++) {
     D[0][j + 1] = INF;
     D[1][j + 1] = j;
  }

//последнее вхождение текущего рассматриваемого символа строки a в  строке b
  int last;
//последнее вхождение каждого символа в строке a
  int lastPosition[SYMBOL_COUNT] = {0};
//индексы, указывающие, что операция транспозиции происходит между
символами a[i] = b[j1] и a[i1] = b[j];
int i1, j1;

  for(int i = 1; i <= n; i++) {
     last = 0;
     for(int j = 1; j <= m; j++)   {
        j1 = last;
        i1 = lastPosition[(int) b[j - 1]];

        if(a[i - 1] == b[j - 1]) {
           D[i + 1][j + 1] = D[i][j];
           last = j;
        }
        else {
           D[i + 1][j + 1] = min(D[i][j + 1] + 1, D[i + 1][j] + 1);
           D[i + 1][j + 1] = min(D[i][j] + 1, D[i + 1][j + 1]);
        }
        // (i - i1 - 1) + 1 + (j - j1 - 1) - стоимость применения транспозиции для
текущего символа
        // если до этого в строке b не встречался символ a[i - 1], то D[i1][j1]
обращает выражение в INF
        D[i + 1][j + 1] = min(D[i1][j1] + (i - i1 - 1) + 1 + (j - j1 - 1), D[i + 1][j + 1]);
     }
     lastPosition[(int)a[i - 1]] = i; //после обработки символа строки a, его индекс
обновляется в массиве lastPosition
  }
  return D[n + 1][m + 1];
}
```

4. Рекурсивный алгоритм

```
//Рекурсивная реализация алгоритма Левенштейна
int DistanseRecursive(const string &a, int i, const string &b, int j)
{
  if(i == 0)
```

```
    return j;
  if(j == 0)
    return i;
  int tmp = min(DistanseRecursive(a, i - 1, b, j) , DistanseRecursive(a, i, b, j - 1)) + 1;
    tmp = min(DistanseRecursive(a, i - 1, b, j - 1) + (int)(a[i - 1] != b[j - 1]), tmp);
  return tmp;
}

int FindDistanceRecursive(const string &a, const string &b)
{
  return DistanseRecursive(a, a.size(), b, b.size());
}
```

Примеры работы

Тесты

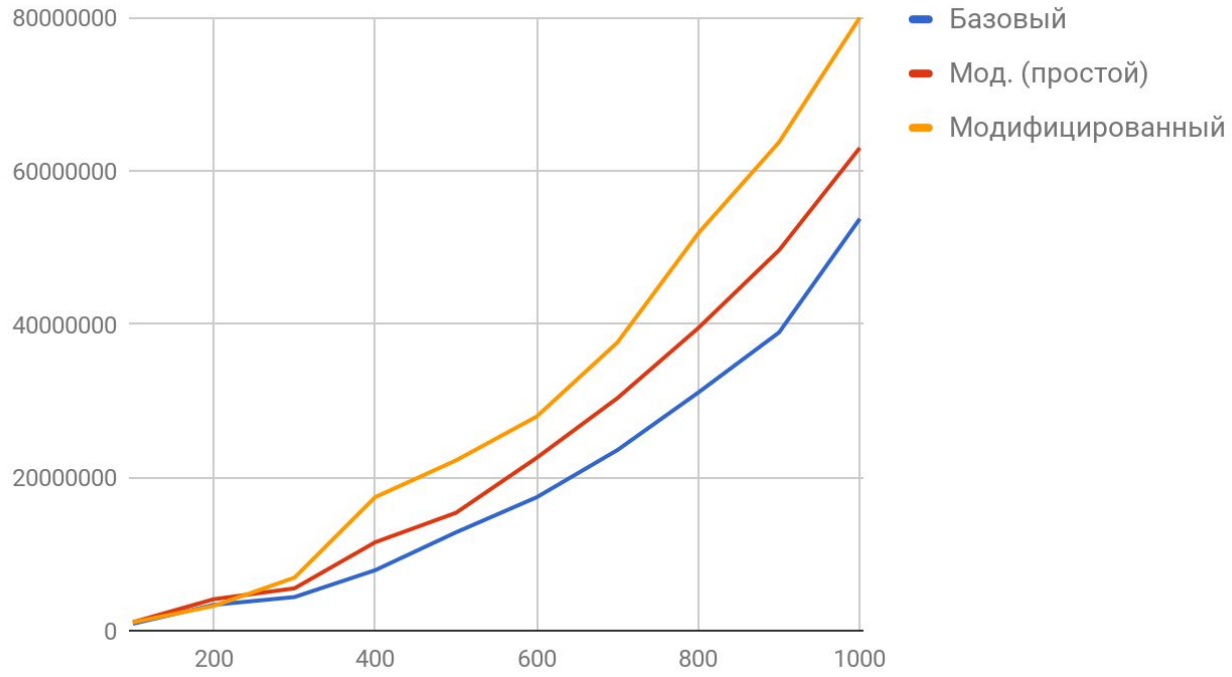
Строка A	Строка B	Результат работы		
		Базовый	Мод. (простой)	Мод. (корректный)
ABCDEFGH	ACDEXGIH	3	3	3
qwerty	qwetry	2	1	1
abcdbikfm	abcibdkfm	2	2	2
abc	ca	3	3	2
matrix	metra	3	3	3
a	b	1	1	1
abcd	bcd	1	1	1
abc	def	3	3	3

Сравнение

Сравнение эффективности работы

Сравнение времени работы

Было произведено сравнение времени работы базового и модифицированных версий алгоритма. Полученные графики подтверждают квадратичную зависимость времени работы алгоритма от длины входных строк.



В данном графике по горизонтали обозначена длина входных строк, по вертикали - среднее количество тиков во время выполнения программы.

Так как в модифицированных алгоритмах используются дополнительные проверка, то их время работы, соответственно выше, чем в базовой реализации.

Сравнение памяти

Базовый и модифицированные алгоритмы работают с помощью построения дополнительной матрицы. Для строк A[n], B[m] алгоритм использует матрицу размера n*m. Соответственно, общие затраты памяти: O(nm). Также, модифицированный алгоритм требует хранить индексы вхождений для каждого символа, которые могут быть поданы на вход в строках.

Рекурсивный алгоритм использует память за счет стека вызовов функций. Так как спуск в рекурсию осуществляется для каждой подстроки входных строк, затрачиваемое количество памяти и времени для этого алгоритма значительно превышает показатели работы нерекурсивной версии алгоритма.

Заключение

Во время выполнения работы был изучен и реализован алгоритм Левенштейна поиска редакционного расстояния между двумя строками с помощью матриц. Была осуществлена реализация базовой версии алгоритма, в том числе и с помощью рекурсии, а также, модифицированной условием добавления операции перестановки элементов. Произведено сравнение реализованных алгоритмов.