

Санкт-Петербургский Государственный Университет

Факультет Прикладной математики – процессов управления

Курсовая работа

по математической логике и теории алгоритмов

на тему:

«Оценка различия текстовых строк»

Выполнила:
Забелина Татьяна Сергеевна,
студентка группы 231

Проверил:
Орлов Вячеслав Борисович

Санкт-Петербург, 2015

Содержание

Введение	2
Постановка задачи	2
Решение задачи	3
1 Расстояния Левенштейна (основные определения)	3
1.1 Формула для вычисления расстояния Левенштейна	4
1.2 Доказательство справедливости формулы	5
2 Алгоритм Вагнера - Фишера	6
3 Нечеткий поиск	8
4 Алгоритм расширения выборки	8
5 Метод N-грамм	10
6 Программная реализация	10
6.1 Описание	10
6.2 Код программы	12
6.3 Примеры работы программы	18
Заключение	19
Список использованной литературы	20

Введение

Очень часто в процессе написания текстов или ввода данных человек допускает ошибки или же опечатки. Большинство текстовых редакторов оснащены системами проверки орфографии, которые считывает введенное слово и ищут идентичное в некотором имеющемся словаре. В случае же опечаток системы оповещают нас об этом, находя *похожие* слова. Здесь-то и появляется надобность получения данных о том, насколько две строки отличаются друг от друга. Аналогичные задачи решают поисковые системы, реализуя функции наподобие «Возможно, Вы имели в виду ...».

В работе рассматриваются алгоритмы вычисления степени различия двух строк, которая играет главную роль в названных выше методах.

В работе не различаются понятия «слово» и «текстовая строка» (или просто «строка»), так как в любом живом языке слово является частью строки (короткой строкой). И словом, и строкой будем называть произвольную последовательность символов некоторого алфавита. Таким образом, будем рассматривать алгоритмы, корректные для любых алфавитов.

Результатом исследования алгоритмов является реализация алгоритма Вагнера-Фишера на языке программирования Java, представленная в последней части работы.

Постановка задачи

Задан некоторый словарь русских имен. Требуется определить, входит ли слово, введенное пользователем, в данный словарь. Если не входит, подсчитать, сколько элементарных коррекций (ЭК) нужно выполнить, чтобы исправить слово. Под ЭК понимаются вставка, замена и исключение символа.

Решение задачи

1 Расстояния Левенштейна (основные определения)

Впервые подобную задачу упомянул в 1965 году советский математик Владимир Иосифович Левенштейн при изучении последовательностей 0-1. Впоследствии более общую задачу для произвольного алфавита, решаемую в данной работе, связали с его именем.[1][2]

Расстояние Левенштейна (также редакционное расстояние или дистанция редактирования) между двумя строками в теории информации и компьютерной лингвистике – это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. Например, расстояние между словами **kitten** и **sitting** равно трем. Этот, и похожие алгоритмы используется 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, в биоинформатике для сравнения генов, хромосом и белков.

Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки второй кратчайшим образом. Обычно действия обозначаются так:

- D (англ. *delete*) – удалить,
- I (англ. *insert*) – вставить,
- R (*replace*) – заменить,
- M (*match*) – совпадение.

Например, для 2-х строк «CONNECT» и «CONEHEAD» можно построить следующую таблицу преобразований:

M	M	M	R	R	R	R	I
C	O	N	N	E	C	T	
C	O	N	E	H	E	A	D

Найти только расстояние Левенштейна – более простая задача, чем найти еще и редакционное предписание.

В случае, когда цены операций (англ. *worth*) могут зависеть от вида операции (вставка, удаление, замена) и/или от участвующих в ней символов, необходимо найти последовательность замен, минимизирующую суммарную цену. В общем случае:

- $w(a, b)$ – цена замены символа a на символ b ,
- $w(\varepsilon, b)$ – цена вставки символа b ,

- $w(a, \varepsilon)$ – цена удаления символа a .

Расстояние Левенштейна является частным случаем этой задачи при

- $w(a,) = 0$,
- $w(a, b) = 1 a \neq b$,
- $w(\varepsilon, b) = 1$,
- $w(a, \varepsilon) = 1$.

Как частный случай, так и задачу для произвольных w , решает *алгоритм Вагнера – Фишера*[1], приведенный ниже. Здесь и далее будем считать, что все w неотрицательны, и действует правило треугольника: если две последовательные операции можно заменить одной, это не ухудшает общую цену (например, заменить символ x на y , а потом y на z не лучше, чем сразу x на z).

1.1 Формула для вычисления расстояния Левенштейна

Для поиска расстояния Левенштейна между двумя строками была выведена ниже описанная и доказанная формула. [1][2]

Здесь и далее считается, что элементы строк нумеруются с первого, как принято в математике и как это делал в своих трудах В. И. Левенштейн, а не с нулевого, как принято во многих языках программирования.

Пусть S_1 и S_2 – две известные строки (длиной M и N соответственно) над некоторым алфавитом. Тогда для решения задачи об их сравнении редакционное расстояние (расстояние Левенштейна) $d(S_1, S_2)$ можно подсчитать по следующей рекуррентной формуле

$d(S_1, S_2) = D(M, N)$, где

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

$$\text{где } m(a, b) = \begin{cases} 0 & ; a = b \\ 1 & ; a \neq b \end{cases},$$

$\min(a, b, c)$ возвращает наименьший из аргументов.

Здесь шаг по i символизирует удаление (D) из первой строки, по j – вставку (I) в первую строку, а шаг по обоим индексам символизирует замену символа (R) или отсутствие изменений (M).

Обозначим длину строки S_1 как $|S_1|$. Тогда очевидна справедливость следующих утверждений:

- $d(S_1, S_2) \geq ||S_1| - |S_2||$
- $d(S_1, S_2) \leq \max(|S_1|, |S_2|)$
- $d(S_1, S_2) = 0 \Leftrightarrow S_1 = S_2$

1.2 Доказательство справедливости формулы

Рассмотрим доказательство приведенной формулы [1].

Очевидно, что редакционное расстояние между двумя пустыми строками равно нулю. Так же очевидно то, что чтобы получить пустую строку из строки длиной i , нужно совершить i операций удаления, а чтобы получить строку длиной j из пустой, нужно произвести j операций вставки. Осталось рассмотреть нетривиальный случай, когда обе строки непусты.

Для начала заметим, что в оптимальной последовательности операций, их можно произвольно менять местами. В самом деле, рассмотрим две последовательные операции:

- Две замены одного и того же символа – неоптимально (если мы заменили x на y , потом y на z , выгоднее было сразу заменить x на z).
- Две замены разных символов можно менять местами
- Два стирания или две вставки можно менять местами
- Вставка символа с его последующим стиранием – неоптимально (можно их обе отменить)
- Стирание и вставку разных символов можно менять местами
- Вставка символа с его последующей заменой – неоптимально (излишняя замена)
- Вставка символа и замена другого символа меняются местами
- Замена символа с его последующим стиранием – неоптимально (излишняя замена)
- Стирание символа и замена другого символа меняются местами

Пусть S_1 заканчивается на символ a , S_2 заканчивается на символ b . Есть три варианта:

1. Символ a , на который заканчивается S_1 , в какой-то момент был стерт. Сделаем это стирание первой операцией. Тогда мы стерли символ a , после чего превратили первые $i - 1$ символов S_1 в S_2 (на что потребовалось $D(i-1, j)$ операций), значит, всего потребовалось $D(i-1, j) + 1$ операций
2. Символ b , на который заканчивается S_2 , в какой-то момент был добавлен. Сделаем это добавление последней операцией. Мы превратили S_1

в первые $j-1$ символов S_2 , после чего добавили b . Аналогично предыдущему случаю, потребовалось $D(i, j - 1) + 1$ операций.

3. Оба предыдущих утверждения неверны. Если мы добавляли символы справа от финального a , то, чтобы сделать последним символом b , мы должны были или в какой-то момент добавить его (но тогда утверждение 2 было бы верно), либо заменить на него один из этих добавленных символов (что тоже невозможно, потому что добавление символа с его последующей заменой – неоптимально). Значит, символов справа от финального a мы не добавляли. Самого финального a мы не стирали, поскольку утверждение 1 неверно. Значит, единственный способ изменения последнего символа – его замена. Заменять его 2 или больше раз неоптимально. Значит,
 - (a) если $a = b$, мы последний символ не меняли. Поскольку мы его также не стирали и не приписывали ничего справа от него, он не влиял на наши действия, и, значит, мы выполнили $D(i - 1, j - 1)$ операций.
 - (b) если $a \neq b$, мы последний символ меняли один раз. Сделаем эту замену первой. В дальнейшем, аналогично предыдущему случаю, мы должны выполнить $D(i - 1, j - 1)$ операций, значит, всего потребуется $D(i - 1, j - 1) + 1$ операций.

2 Алгоритм Вагнера - Фишера

Этот алгоритм предложен Р. Вагнером (R. A. Wagner) и М. Фишером (M. J. Fischer) в 1974 году. [1][3]

Алгоритм решает задачу нахождения расстояния Левенштейна в общем случае, используя вышеприведенную формулу. Кроме того он позволяет восстановить редакционное предписание. Для этого вычисляется матрица расстояний размерностью $[M \times N]$. Элемент матрицы, соответствующий i -ой строке и j -ому столбцу, равен $D(i, j)$. Ее можно вычислять как по строкам, так и по столбцам.

Псевдокод ниже демонстрирует работу алгоритма Вагнера-Фишера для

произвольных цен операций (замены, вставки и удаления).

```

 $D(0, 0) = 0;$ 
for  $j = 1 \dots N$  do
     $| D(0, j) = D(0, j - 1) + \text{цена вставки символа } S_2[j];$ 
end
for  $i = 1 \dots M$  do
     $| D(i, 0) = D(i - 1, 0) + \text{цена удаления символа } S_1[i];$ 
    for  $j = 1 \dots N$  do
         $| D(i, j) = \min(D(i - 1, j) + \text{цена удаления символа } S_1[i],$ 
         $| D(i, j - 1) + \text{цена вставки символа } S_2[j],$ 
         $| D(i - 1, j - 1) + \text{цена замены символа } S_1[i] \text{ на символ } S_2[j],$ 
         $| );$ 
    end
end
вернуть  $D(M, N)$ 
```

Для восстановления редакционного предписания требуется проанализировать матрицу D : будем идти из правого нижнего угла (M, N) в левый верхний, на каждом шаге ища минимальное из трех значений:

- если минимально $(D(i-1, j) + \text{цена удаления символа } S_1[i])$, добавляем удаление символа $S_1[i]$ и идем в $(i - 1, j)$
- если минимально $(D(i, j-1) + \text{цена вставки символа } S_2[j])$, добавляем вставку символа $S_2[j]$ и идем в $(i, j - 1)$
- если минимально $(D(i-1, j-1) + \text{цена замены символа } S_1[i] \text{ на символ } S_2[j])$, добавляем замену $S_1[i]$ на $S_2[j]$ (если они не равны; иначе ничего не добавляем), после чего идем в $(i - 1, j - 1)$

Здесь (i, j) – клетка матрицы, в которой мы находимся на данном шаге. Если минимальны два из трех значений (или равны все три), это означает, что есть 2 или 3 равноценных редакционных предписания.

Пример восстановления редакционного предписания для слов *CONNECT* и *CONEHEAD* по матрице расстояний

	<i>C</i>	<i>O</i>	<i>N</i>	<i>N</i>	<i>E</i>	<i>C</i>	<i>T</i>
<i>C</i>	0	1	2	3	4	5	6
<i>O</i>	1	0	1	2	3	4	5
<i>N</i>	2	1	0	1	2	3	4
<i>E</i>	3	2	1	1	1	2	3
<i>H</i>	4	3	2	2	2	2	3
<i>E</i>	5	4	3	3	2	3	3
<i>A</i>	6	5	4	4	3	3	4
<i>D</i>	7	6	5	5	4	4	4

Алгоритм в виде, описанном выше, требует $O(M \cdot N)$ операций и такую же память. Последнее может быть неприятным: так, для сравнения файлов длиной в 10^5 строк потребуется около 40 Гб памяти.

Если требуется только расстояние, легко уменьшить требуемую память до $O(\min(M, N))$. Для этого надо учесть, что после вычисления любой строки предыдущая строка больше не нужна. Более того, после вычисления $D(i, j)$ не нужны также $D(i - 1, 0), \dots, D(i - 1, j - 1)$.

3 Нечеткий поиск

Обобщенное сопоставления строк, включающая в себя нахождение подстрок строки, близких к заданному образцу строки, называется нечетким сопоставлением строк или *нечетким поиском*.

Задачу нечеткого поиска можно сформулировать следующим образом: «По заданному слову найти в тексте или словаре размера n все слова, совпадающие с этим словом (или начинающиеся с этого слова) с учетом k возможных различий».

Нечеткий поиск является крайне полезной функцией любой поисковой системы. Алгоритмы нечеткого поиска являются основой систем проверки орфографии и полноценных поисковых систем вроде Google или Yandex. Вместе с тем, его эффективная реализация намного сложнее, чем реализация простого поиска по точному совпадению.^[4]

Алгоритмы нечеткого поиска характеризуются метрикой – функцией расстояния между двумя словами, позволяющей оценить степень их сходства в данном контексте. В числе наиболее известных метрик – расстояние Левенштейна.

Далее в работе рассмотрены некоторые алгоритмы нечеткого поиска, которые потенциально можно использовать в решении поставленной задачи. В основном же алгоритмы нечеткого поиска рассчитаны на более сложные задачи с ограничениями на время работы и используемую память.

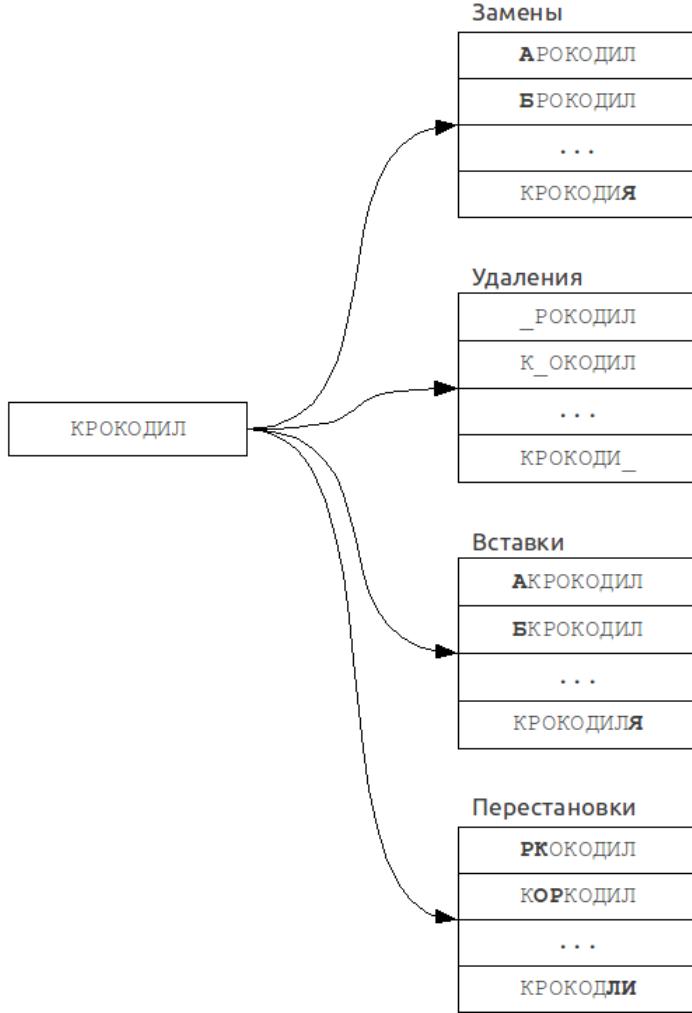
4 Алгоритм расширения выборки

Этот алгоритм часто применяется в системах проверки орфографии, там, где размер словаря невелик, либо же где скорость работы не является основным критерием. Он основан на сведении задачи о нечетком поиске к задаче о точном поиске.

Из исходного запроса строится множество всех «ошибочных» слов, для каждого из которых затем производится точный поиск в словаре.

Время его работы сильно зависит от числа k ошибок и от размера алфавита A , и в случае использования бинарного поиска по словарю составляет:

Рис. 1: Работа алгоритма расширения выборки



$$O((M|A|)^k \cdot M \cdot \log(N))$$

Например, при $k = 1$ и слова длины 7 (например, «Крокодил») в русском алфавите множество ошибочных слов будет размером около 450, то есть будет необходимо сделать 450 запросов к словарю, что вполне приемлемо. Но уже при $k = 2$ размер такого множества будет составлять более 115 тысяч вариантов, что соответствует полному перебору небольшого словаря, и, следовательно, время работы будет достаточно велико. При этом не нужно забывать еще и о том, что для каждого из таких слов необходимо провести поиск на точное совпадение в словаре.

Алгоритм может быть легко модифицирован для генерации «ошибочных» вариантов по произвольным правилам, и, к тому же, не требует никакой предварительной обработки словаря, и, соответственно, дополнительной памяти.

Можно генерировать не все множество «ошибочных» слов, а только те из них, которые наиболее вероятно могут встретиться в реальной ситуации, например, слова с учетом распространенных орфографических ошибок или ошибок набора.

5 Метод N-грамм

Этот метод был придуман довольно давно, и является наиболее широко используемым, так как его реализация крайне проста, и он обеспечивает достаточно хорошую производительность. Алгоритм основывается на принципе: «Если одно слово совпадает с другим словом с учетом нескольких ошибок, то с большой долей вероятности у них будет хотя бы одна общая подстрока длины N ». Эти подстроки длины N называются N -граммами. Во время индексации слово разбивается на такие N -граммы, а затем это слово попадает в списки для каждой из этих N -грамм. Во время поиска запрос также разбивается на N -граммы, и для каждой из них производится последовательный перебор списка слов, содержащих такую подстроку.

Наиболее часто используемыми на практике являются триграммы – подстроки длины 3. Выбор большего значения N ведет к ограничению на минимальную длину слова, при которой уже возможно обнаружение ошибок.

Алгоритм N -грамм находит не все возможные слова с ошибками. Возьмем, например, слово ЛОТКА, и разложим его на триграммы:

ЛОТКА → ЛОТ ОТК ТКА.

Можно заметить, что они все содержат ошибку Т. Таким образом, слово «ЛОДКА» найдено не будет, так как оно не содержит ни одной из этих триграмм, и не попадет в соответствующие им списки. Таким образом, чем меньше длина слова и чем больше в нем ошибок, тем выше шанс того, что оно не попадет в соответствующие N -граммам запроса списки, и не будет присутствовать в результате.

Между тем, метод N -грамм оставляет полный простор для использования собственных метрик с произвольными свойствами и сложностью, но за это приходится платить – при его использовании остается необходимость в последовательном переборе около 15% словаря, что достаточно много для словарей большого объема.

6 Программная реализация

6.1 Описание

Для программной реализации поставленной задачи был выбран алгоритм Вагнера-Фишера. В сравнении с другими рассмотренными в работе алгоритмами он быстро производит поиск несоответствий в строках без лишнего использования памяти. В качестве метрики для сравнения строк используется расстояние Левенштейна.

При реализации алгоритма было учтено, что в языке программирования Java элементы строк и массивов нумеруются с нулевого, а не с первого, как

это принималось в теоретической части работы.

В представленной ниже программе создается «словарь» русских имен, в которыйчитываются данные (имена) из текстового файла. Список имен был взят из открытого источника в интернете для решения поставленной задачи и не является истинным набором всех русских имен [5]. После запуска программы пользователю предлагается ввести имя. Введенное слово ищется в словаре. Выводится информация о том, было ли найдено пользовательское слово в словаре. В случае отсутствия слова в словаре создается и демонстрируется массив имен, наиболее похожих на введенное пользователем слово. Осуществляется также поиск редакционного предписания для каждого из найденных слов. Программа завершает свою работу после того, как в словаре было в точности найдено введенное слово. До этих пор пользователю предлагается произвести ввод данных.

Вводятся могут строки из любых символов, так как сравнение строк в конечном итоге производится посимвольно. Если в введенной строке присутствуют символы, не входящие в кириллицу, программа примет такую строку и выведет наиболее похожее имя из словаря, не содержащее некорректных символов, а также выдаст сообщение об опечатках. Время работы алгоритма $T = O(\sum_{i=1}^N m \cdot n_i)$, где N - количество слов в словаре, n_i - длина i -ого слова в словаре и m - длина введенной строки. Программа рассчитана на ввод русских имен, а они обычно не превышают 15 символов. Поэтому такое время весьма приемлемо и пользователю практически не приходится ожидать результатов работы программы. Если же ввести довольно длинную строку, то алгоритм отработает корректно, но достаточно долго. Во избежание чрезмерно долгой работы программа оснащена проверкой на длину введенной строки. Если она больше чем на 3 превышает длину максимального слова словаря, то пользователю выводится предупреждение и предлагается уменьшить длину введенного слова или продолжить вычисления. Таким образом допускается не более трех лишних символов при вводе слова максимальной длины. Длина максимального слова вычисляется непосредственно в программе после загрузки в память массива слов из словаря.

6.2 Код программы

```
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {

        Dictionary names = new Dictionary(); //создание словаря русских имен
        names.print(); //вывод на экран имеющихся в
словаре слов
        Scanner sc = new Scanner(System.in);
        String userName;
        boolean isFound = false;
        enter: do { //цикл выполняется пока не
будет введено слово, имеющееся в словаре
            System.out.print("\nВведите имя: ");
            userName = sc.next(); //считывание ввода
            //преобразование к правильному регистру и вывод на экран
            System.out.println("Вы ввели " + userName.substring(0, 1).toUpperCase() +
userName.substring(1).toLowerCase() + "\n");
            // проверка на длину слова
            if(userName.length() >= names.MAX_LENGTH + 3){ // допускается
3 лишних символа в словах максимальной длины
                System.out.println("В словаре нет слов такой длины. Рекомендуем
уменьшить длину слова в запросе. ");
                System.out.println("Работа с данным словом может занять много времени.
Продолжить работу с данным словом? (д/н) ");
                String answer = sc.next();
                do{
                    if (answer.equalsIgnoreCase("н")){ // если нет,
то продолжаем цикл по вводу слова
                        continue enter;
                    }
                    else if(!answer.equalsIgnoreCase("д")){ // если не да,
то продолжаем цикл по вводу ответа
                        System.out.println("Работа с данным словом может занять
много времени. Продолжить работу с данным словом? (д/н) ");
                        answer = sc.next();
                    }
                    else break;
                } // да, тогда выходим из цикла и ищем слово
            } while(true);
        }
        System.out.println("Поиск идет...");
        isFound = names.search(userName); //только здесь запускается поиск
по словарю, а в нем и все необходимые методы
    }while(!isFound);
    System.out.println("Программа завершила работу.");
}
}
```

```

import java.util.*;
import java.io.*;

public class Dictionary {
    public List<String> list;           // массив для словаря имен
    public int MAX_LENGTH;               // длина самого длинного слова

    // конструктор
    public Dictionary() {
        list = new ArrayList<String>();
        BufferedReader rd = null;          // для чтения из файла
        try{
            // Открытие файла для чтения
            rd = new BufferedReader(new InputStreamReader(new
FileInputStream("nameList.txt"), "windows-1251"));
            // Чтение содержимого файла
            while((rd.readLine()) != null){
                list.add(rd.readLine());
            }
        }
        catch(IOException e){           // ловим исключение на случай, если файл не
найден
            System.err.print("An IOException was caught!");
            e.printStackTrace();
        }
        finally{
            // Закрытие файла
            try{
                rd.close();
            }
            catch(IOException e){
                System.err.print("An IOException was caught!");
                e.printStackTrace();
            }
        }
        // считаем максимальную длину слова
        this.MAX_LENGTH = this.list.get(0).length();
        for(String eachWord : this.list){
            this.MAX_LENGTH = eachWord.length();
        }
    }
    System.out.println("Информация о словаре\nМаксимальная длина слова: " +
this.MAX_LENGTH + "\nВсего слов: " + this.list.size());
}

// добавляет новые слова и восстанавливает алфавитный порядок
public void push(String... newWords) {
    // цикл по каждому введенному слову
    newWordsLoop : for(String eachWord : newWords) {
        // цикл по каждому слову в словаре
        for (String eachDictWord : this.list) {
            if (eachWord.equalsIgnoreCase(eachDictWord)) {           // если такое
слово уже есть в словаре
                continue newWordsLoop;                                // -- не
добавляем
            }
        }
        //иначе добавляем, устанавливая правильный регистр (с большой буквы)
    }
}

```

```

        this.list.add(eachWord.substring(0, 1).toUpperCase() +
eachWord.substring(1).toLowerCase()));

    }
    this.sort();           // восстанавливаем алфавитный порядок
}

public void print() {                                // вывод всех слов в столбик
    System.out.println("Словарь:\n");
    // цикл по каждому слову в массиве слов
    for (String eachWord : this.list) {
        System.out.println(eachWord);
    }
}

public void sort() {                                // сортирует массив по
алфавиту
    Collections.sort(list, new Comparator<String>() {
        public int compare(String str1, String str2) {
            return str1.compareTo(str2);
        }
    });
}

public boolean search(String userName) {             // поиск слова в словаре, в
случае его отсутствия поиск наиболее похожего
    for(String eachWord : this.list){
        if(eachWord.equalsIgnoreCase(userName)) {      // слово есть в словаре
            System.out.println("Такое имя есть в словаре: " + eachWord);
            return true;
        }
    }
}

userName = userName.toUpperCase();                  // приведение к верхнему
регистру -- для сравнение без учета регистра

// массив слов с наименьшим расстоянием Левенштейна
List<String> appropriate = new ArrayList<String>();
// пусть наиболее похожее слово -- первое
appropriate.add(0, this.list.get(0));
// наименьшее расстояние Левенштейна
int appropriateDistance = distanceLevenstein(appropriate.get(0).length(),
userName.length(), appropriate.get(0).toUpperCase(), userName);
for (String eachWord : this.list) {                // проверяем каждое слово из
словаря и ищем наименьшее расстояние
    int eachDist = distanceLevenstein(eachWord.length(), userName.length(),
eachWord.toUpperCase(), userName);
    // если нашли новый минимум --
    if(eachDist < appropriateDistance) {
        // -- то очищаем массив похожих слов
        appropriate.clear();
        // -- добавляем в массив наиболее похожее слово
        appropriate.add(eachWord);
        appropriateDistance = eachDist;
    }
    // если равно минимальному --
    else if(eachDist == appropriateDistance){
        // -- то добавляем новое слово в массив
    }
}

```

```

                appropriate.add(eachWord);
            }
        }
        // вывод на экран похожих слов
        System.out.println("Слово не найдено в словаре.\nВозможно, Вы имели в виду:");
    };
    for(String eachWord : appropriate){
        System.out.println(eachWord);
    }
    System.out.println("Расстояние Левенштейна: " + appropriateDistance + "\n");

    // расчет матрицы и редакционного предписания
    for(String eachWord : appropriate){
        setDistMatrix(userName, eachWord.toUpperCase());
    }
    return false; //слово не найдено
}

public int distanceLevenstein(int i, int j, String s1, String s2) { // возвращает расстояние Левенштейна -- в тексте D(M, N)
    if (j == 0) return i; // вторая строка пустая, в том числе обе строки пустые
    if (i == 0) return j; // первая строка пустая

    byte match; // хранит факт совпадения символов
    if (s1.charAt(i - 1) == s2.charAt(j - 1)) { // сравнение символов
        match = 0;
    } else {
        match = 1;
    }
    // выбор случая с минимальным расстоянием
    return minimum(distanceLevenstein(i - 1, j, s1, s2) + 1, // удаление символа
                   distanceLevenstein(i, j - 1, s1, s2) + 1, // добавление символа
                   distanceLevenstein(i - 1, j - 1, s1, s2) + match); // замена символа в случае неидентичности
}
// минимум из множества целых чисел
public int minimum(int...x){
    int min = x[0];
    for(int eachValue : x){ // цикл выполняется для каждого значения x[i]
        if (eachValue < min) min = eachValue;
    }
    return min;
}

public void setDistMatrix(String userName, String dictName) { // заполнение матрицы расстояний
    System.out.println(userName + " -- " + dictName + "\nМатрица расстояний Левенштейна:");
    //создание матрицы
    int[][][] matrix = new int[userName.length() + 1][dictName.length() + 1];
    for (int i = 0; i <= userName.length(); i++) {
        for (int j = 0; j <= dictName.length(); j++) {
            matrix[i][j] = distanceLevenstein(i, j, userName, dictName);
            System.out.print(matrix[i][j] + "\t");
        }
    }
    System.out.println();
}

```

```

    }
    // восстановление редакционного предписания по вычисленной матрице
    editDirection(userName, dictName, matrix);
}

public void editDirection(String userName, String dictName, int[][][] matrix){
// восстановление редакционного предписания

    System.out.println("Восстановление редакционного предписания:" + "\n" +
userName);
    // идем с правого нижнего угла в левый верхний
    int i = userName.length();
    int j = dictName.length();
    while (i > 0 || j > 0) {           // пока находимся в пределах матрицы
        if(i == 0){                  // находимся в крайней верхней строке --
        может идти только влево
            userName = dictName.charAt(j - 1) + userName;           // вставили
        символ вначало
            System.out.println(userName + "\t// вставили символ " +
dictName.charAt(j - 1));
            j--;
        }
        else if(j == 0){           // находимся в крайнем левом столбце -- можем
        идти только вверх
            char deleted = userName.charAt(i - 1);           // запоминаем
        символ, который будем удалять, чтобы вывести позже на экран
            userName = userName.substring(0, i - 1) + userName.substring(i,
userName.length());           // удалили символ
            System.out.println(userName + "\t// удалили символ " + deleted);
            i--;
        }
        // идем по диагонали
        else if (minimum(matrix[i - 1][j], matrix[i][j - 1], matrix[i - 1][j - 1]) ==
matrix[i - 1][j - 1]) {
            if (userName.charAt(i - 1) != dictName.charAt(j - 1)) {           // если
        символы не равны, то делаем замену, иначе - пропускаем
                char deleted = userName.charAt(i - 1);           // запоминаем
        символ, который будем удалять, чтобы вывести позже на экран
                userName = userName.substring(0, i - 1) + dictName.charAt(j - 1)
+ userName.substring(i, userName.length());           // заменили символ
                System.out.println(userName + "\t// заменили символ " + deleted +
" на " + dictName.charAt(j - 1));
            }
            i--;
            j--;
        }
        // идем наверх
        else if (minimum(matrix[i - 1][j], matrix[i][j - 1], matrix[i - 1][j - 1]) ==
matrix[i - 1][j]) {
            char deleted = userName.charAt(i - 1);           // запоминаем
        символ, который будем удалять, чтобы вывести позже на экран
            userName = userName.substring(0, i - 1) + userName.substring(i,
userName.length());           // удалили символ
            System.out.println(userName + "\t// удалили символ " + deleted);
            i--;
        }
        // идем влево
        else {
            userName = userName.substring(0, i) + dictName.charAt(j - 1) +
userName.substring(i, userName.length());           // вставили символ
            System.out.println(userName + "\t// вставили символ " +

```

```
dictName.charAt(j - 1));
                j--;
            }
        }

        // проверка на совпадение полученных слов
        if(userName.equals(dictName)){
            return;
        }

        else{
            System.out.println("Ошибка! Получились неодинаковые слова");
        }
    }
}
```

6.3 Примеры работы программы

На фотографиях ниже продемонстрирована работа программы в двух возможных случаях в зависимости от того, имеется ли введенное слово в словаре. Тестовыми словами в данных случаях были "татьяна" и "алексей" соответственно.

Филипп
Фома
Цветана
Юлия
Юрий
Яна
Ярина
Ярослава

Введите имя: татьяна
Вы ввели Татьяна

Поиск идет...
Такое имя есть в словаре: Татьяна
Программа завершила работу.

Введите имя: алексей
Вы ввели Алексей

Поиск идет...
Слово не найдено в словаре.
Возможно, Вы имели в виду:
Елисей
Расстояние Левенштейна: 3

АЛЕКСЕЙ -- ЕЛИСЕЙ

Матрица расстояний Левенштейна:

0	1	2	3	4	5	6
1	1	2	3	4	5	6
2	2	1	2	3	4	5
3	2	2	2	3	3	4
4	3	3	3	3	4	4
5	4	4	4	3	4	5
6	5	5	5	4	3	4
7	6	6	6	5	4	3

Восстановление редакционного предписания:

АЛЕКСЕЙ
АЛЕИСЕЙ // заменили символ К на И
АЛИСЕЙ // удалили символ Е
ЕЛИСЕЙ // заменили символ А на Е

Введите имя: Елисей
Вы ввели Елисей

Поиск идет...
Такое имя есть в словаре: Елисей
Программа завершила работу.

На данных примерах видно, что в программе слова сравниваются без

учета регистра, а также приводятся к регистру собственных имен (с большой буквы).

В случае, когда введенное слово в словаре не найдено, пользователю предлагаются *наиболее похожие* (с минимальным редакционным расстоянием). Также выводится на экран матрица расстояний, по которой далее составлено редакционное предписание.

После выведенной информации пользователю предоставляется возможность ввести слово заново.

Заключение

В ходе работы решена задача о поиске русских имен, наиболее похожих на введенное слово или имя. Для решения данной задачи были рассмотрены алгоритмы для вычисления степени различия двух текстовых строк. Под текстовой строкой или словом в работе понимается произвольная последовательность символов некоторого алфавита.

Результатом теоретических исследований стала программная реализация алгоритма Вагнера-Фишера, который успешно решает поставленную задачу. Этот алгоритм был выбран, так как он достаточно быстрый и не излишне требовательный к памяти в условиях рассматриваемой задачи. Алгоритм Вагнера-Фишера реализован на языке программирования Java.

В программе используется список русских имен, взятый из открытого источника в интернете. Список имен составляет 117 слов. Введенное пользователем слово ищется в словаре. В случае отсутствия этого слова в словаре пользователю демонстрируются слова, наиболее похожие на введенное пользователем. Демонстрируется также восстановленное редакционное предписание для каждого из найденных слов.

Программа корректно отработает при любых входных символах и при достаточно большой длине введенной строки. Если в введенной строке присутствуют символы, не входящие в кириллицу, программа примет такую строку и выведет наиболее похожее имя из словаря, не содержащее некорректных символов, а также выдаст сообщение об опечатках. Во избежание чрезмерно долгой работы программа оснащена проверкой на длину введенной строки. Если она больше чем на 3 превышает длину максимального слова словаря, то пользователю выводится предупреждение и предлагается уменьшить длину введенного слова или продолжить вычисления. Длина максимального слова вычисляется непосредственно в программе после загрузки в память массива слов из словаря.

Написанная программа может быть использована, например, в качестве функциональной части системы проверки орфографии в некотором текстовом редакторе или поисковой системе. В этом случае программу следует отредактировать так, чтобы пользователь был уведомлен о правильности написания слова (часто это делается подчеркиванием слова в случае нахождения в нем ошибки). А также пользователю необходимо дать подсказки по исправлению слова – вывести наиболее похожие слова. Для этого достаточно лишь не выводить на экран избыточную информацию, такую как матрицы расстояний и редакционное предписание.

Список использованной литературы

1. Расстояние Левенштейна, алгоритм Вагнера-Фишера: <https://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D1%81%D1%81%D1%82%D0%BE%D1%8F%D0%BD%D0%B8%D0%B5%D0%9B%D0%B5%D0%B2%D0%B5%D0%BD%D1%88%D1%82%D0%B5%D0%B9%D0%BD%D0%BD>
2. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академии Наук СССР, 1965 163.4:845-848.
3. R. A. Wagner, M. J. Fischer. The string-to-string correction problem. J. ACM 21 1 (1974) P. 168—173.
4. Алгоритмы нечеткого поиска: <http://habrahabr.ru/post/114997/>
5. Список русских имен: <http://www.kakzovut.ru/russkie-imena.html>