МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное

образовательное учреждение высшего образования

**«Челябинский государственный университет»**

**(ФГБОУ ВО «ЧелГУ»)**

Математический факультет

Кафедра теории управления и оптимизации

**КУРСОВАЯ РАБОТА**

Об одной задаче поиска по сходству

Выполнил студент Караблев Антон

Павлович

Группы МП-201

очной формы обучения

направления подготовки

01.03.02 Прикладная математика и информатика \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

«\_\_\_\_» \_\_\_\_\_\_\_\_\_\_\_\_ 2021г.

|  |  |
| --- | --- |
|  | Научный руководитель  Алеева Сюзанна Рифхатовна  Должность доцент  Ученая степень канд. физ.-мат. наук    «\_\_\_» \_\_\_\_\_\_\_\_\_ 2021г.  Научный руководитель  Красильникова Екатерина Сергеевна  Должность старший преподаватель    «\_\_\_» \_\_\_\_\_\_\_\_\_ 2021г. |

Челябинск

2021

**Содержание**

[Введение 3](#_Toc74130777)

[Триграммный индекс 4](#_Toc74130778)

[Методы нечеткого поиска 6](#_Toc74130779)

[ВК-деревья 6](#_Toc74130780)

[Алгоритм расширенной выборки 7](#_Toc74130781)

[Bitap 9](#_Toc74130782)

[Реализация 10](#_Toc74130783)

[Заключение 13](#_Toc74130784)

[Список литературы 14](#_Toc74130785)

[Приложение1 15](#_Toc74130786)

# Введение

Поиск по сходству— это поиск информации, при котором выполняется сопоставление информации заданному образцу поиска или близкому к этому образцу значению. Алгоритмы нечеткого поиска используются в большинстве современных поисковых систем, например, для проверки орфографии, о чем мы с вами дальше и будем говорить.

Нечеткий поиск является крайне полезной функцией любой поисковой системы. Вместе с тем его эффективная реализация намного сложнее, чем реализация простого поиска по точному совпадению.

Задачу нечеткого поиска можно сформулировать так:

«По заданному слову найти в тексте или словаре размера ***n*** все слова, совпадающие с этим словом с учетом ***k*** возможных различий».

Например, при запросе «Машина» с учетом двух возможных ошибок, найти слова «Машинка», «Махина», «Малина», «Калина» и так далее.

Алгоритмы нечеткого поиска характеризуются *метрикой* — функцией расстояния между двумя словами, позволяющей оценить степень их сходства в данном контексте.

# Триграммный индекс

Этот метод был придуман довольно давно, и является наиболее широко используемым, так как его реализация крайне проста, и он обеспечивает достаточно хорошую производительность.

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 основывается на принципе: «Если слово **А** совпадает со словом **Б** с учетом нескольких ошибок, то с большой долей вероятности у них будет хотя бы одна общая подстрока длины *N*». Эти подстроки длины *N* и называются *N*-граммами.

**Метод *N*-грамм состоит из нескольких этапов:**

1. Составляются индексные базы по указанным *N*-граммам.
2. Введенное слово для поиска тоже разбивается на *N*-граммы.
3. Выполняется полный перебор значений по указанным *N*-граммам.

Пример такого метода показан на *рис.1*

**Формула, для расчета метрики**: *P* = (*a* + *b*) / (2 \* *m*), где

* *m* – число совпадающих триграмм
* *a* – общее число триграмм в первой строке
* *b* – общее число триграмм во второй строке

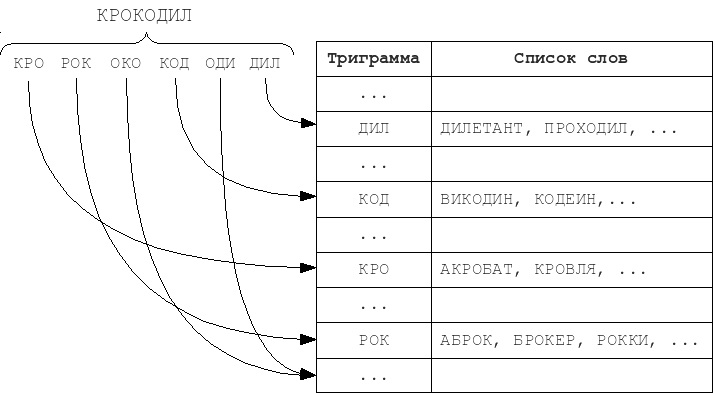
**Пример расчета расстояния с помощью триграмм**:

Строка 1: *ALEXANDRE*

Строка 2: *ALEKSANDER*

*P* = (*a* + *b*) / (2 \* *m*) = (9 + 10) / (2 \* 3) = 3,16

Соответственно, чем меньше будет коэффициент, тем ближе по сходству слова.



*Рис. 1. Поиск ошибок методом N-грамм*

Этот метод не всегда находит ошибки. Данная проблема происходит, например, когда пользователь ошибся в одной букве в корне слова при вводе, тогда соответствующие триграммы корня не будут найдены в словаре и процент нахождения правильно исправленного слова заметно снизится. Так же при наличии словарей больших размеров данный метод будет долго выполняться и требовать больших ресурсов и мощностей.

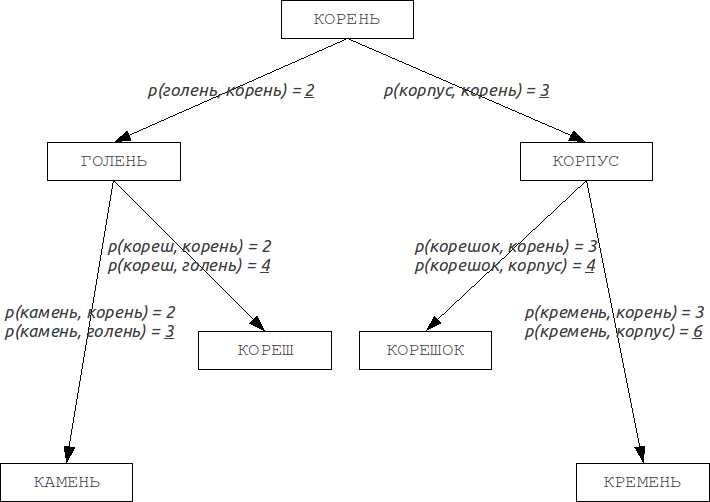
# Методы нечеткого поиска

**ВК-деревья**

Деревья *Burkhard-Keller* являются метрическими деревьями, алгоритмы построения таких деревьев основаны на свойстве метрики отвечать неравенству треугольника:

*p*(*x*, *y*) <= *p*(*x*, *y*) + *p*(*z*, *y*), *x*,*y*,*z* ∈ *X*.

Это свойство позволяет метрикам образовывать метрические пространства произвольной размерности. Такие метрические пространства не обязательно являются евклидовыми. На основании этих свойств можно построить структуру данных, осуществляющую поиск в таком метрическом пространстве, которой и являются деревья *Burkhard-Keller*.

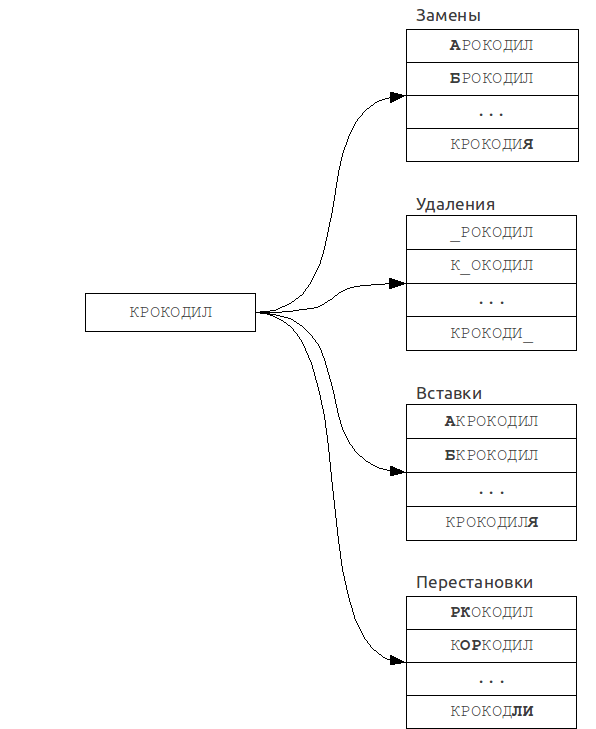


*Рис. 2. Дерево Burkhard-Keller*

**Алгоритм расширенной 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 «ошибочных» слов, для каждого из которых затем производится точный поиск в словаре. Время его работы сильно зависит от числа ошибок и от размера алфавита.



*Рис. 3. Поиск ошибок методом расширения выборки*

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. Так же, можно генерировать не всё 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.

**Bitap**

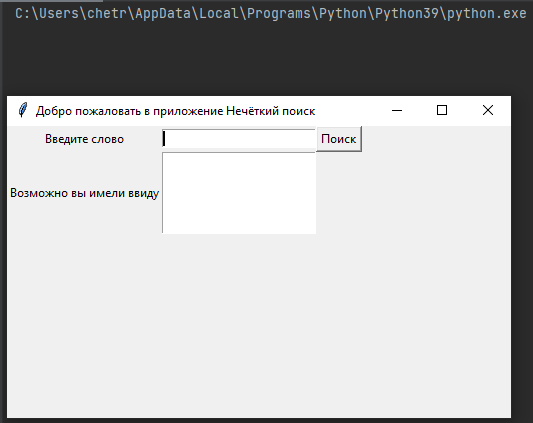
Алгоритм *Bitap* и различные его модификации наиболее часто используются для нечеткого поиска без индексации.

Впервые идею этого алгоритма предложили граждане *Ricardo Baeza*-*Yates* и *Gaston Gonnet*, опубликовав соответствующую статью в 1992 году. Оригинальная версия алгоритма имеет дело только с заменами символов, и, фактически, вычисляет расстояние Хемминга. Но немного позже *Sun Wu* и *Udi Manber* предложили модификацию этого алгоритма для вычисления расстояния Левенштейна, т.е. привнесли поддержку вставок и удалений, и разработали на его основе первую версию утилиты *agrep*.

Совпадение или несовпадение по запросу определяется самым последним битом результирующего вектора. Высокая скорость работы этого алгоритма обеспечивается за счет битового параллелизма вычислений — за одну операцию возможно провести вычисления над 32 и более битами одновременно.

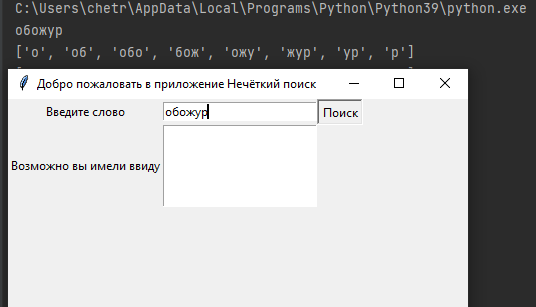
Несмотря на то, что асимптотическое время работы этого алгоритма *O(n)* совпадает с таковым у линейного метода, он значительно быстрее при длинных запросах и количестве ошибок более 2. Очевидно, что простой перебор с использованием метрики, в отличие от алгоритма *Bitap*, сильно зависит от количества ошибок. Тем не менее, если речь заходит о поиске в неизменных текстах большого объема, то время поиска можно значительно сократить, произведя предварительную обработку такого текста, также называемую индексацией.

# Реализация



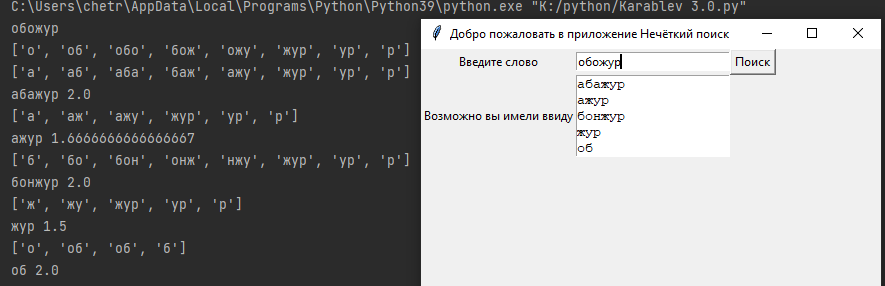
*Рис. 1.1*

На *рис. 1.1* при запуске программы открывается окно импортированной библиотеки *tkinter*, отвечающей за графический дизайн в программе, которая ждет когда в поле для ввода напишут исходное слово с ошибкой и нажмут кнопку “Поиск”.



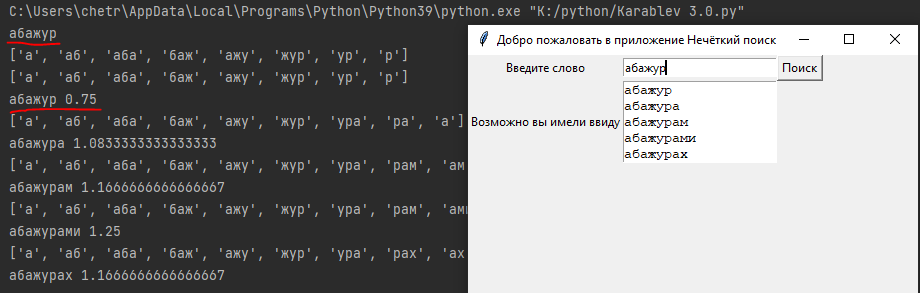
*Рис. 1.2*

На *рис. 1.2* после активации кнопки обрабатывается введенное слово. Оно переходит в нижний регистр, если всё слово или некоторые буквы были заглавными. Так же все символы, кроме русского алфавита и цифр, будут заменены на пустые. Это сделано потому что, программа работает с русским алфавитом.



*Рис. 1.3*

На *рис. 1.3* в консоль выводится само слово из словоря и его триграмма с коэффициентом. Чем меньше коэффициент тем более вероятно что исходное слово и слово из словоря одинаковы в написании. Например, на рис. 1.4 введено слово без ошибки “абажур” и идентичное ему слово из словоря выводится с коэффициентом “0.75”.



*Рис. 1.4*

В программе используется словарь с более чем одним миллионом слов, поэтому для ограничения вывода принято решение выволить пять первых слов удовлетворяющих условию, коэффициент не должен превышать двух.

# Заключение

В курсовой работе были рассмотрены такие методы поиска по сходству как, *ВК-деревья, Bitap*,*Алгоритм расширенной выборки*, а также во всех деталях основная тема курсовой - *триграммный индекс*. Её код вы можете найти в *приложение1*. Существует также еще множество разнообразных методов, основанных, помимо всего прочего, на адаптации различных, уже где-либо применяемых техник и приемов. Таким образом, практическое использование алгоритмов нечеткого поиска в реальных поисковых системах тесно связано с фонетическими алгоритмами.

# Список литературы

1. [сайт]. URL: https://python-scripts.com/tkinter
2. [сайт]. URL: https://pythonworld.ru/tipy-dannyx-v-python/stroki-funkcii-i-metody-strok.html
3. [сайт]. URL: https://coderlessons.com/articles/ruby/udivitelnoe-avtozapolnenie-poisk-trigramm-v-rails-i-postgresql
4. [сайт]. URL: https://m.habr.com/ru/post/114997/
5. [сайт]. URL: https://hmath.spbstu.ru/userfiles/files/documents/conference/metriki\_v\_informatike.pdf
6. [сайт]. URL: https://habr.com/ru/post/78566/

# Приложение1

import re  
  
  
from tkinter import \*  
  
  
window = Tk()  
window.geometry('450x250')  
window.title("Добро пожаловать в приложение Нечёткий поиск")  
  
  
def trigram(length, sought, arr):  
 m = 1  
 a = 0  
 m1 = 0  
 a1 = 0  
 for i in range(0, length + 2):  
 k = sought[i + a + a1:i + m1 + m]  
 if a == -1:  
 m1 = 1  
 a1 = -1  
 m = 0  
 a = -1  
 arr.append(k)

def get\_data():  
 book = open('russian.txt')  
 num\_lines = sum(1 for line in open('russian.txt'))  
 return num\_lines, book  
  
  
def main():

text.delete(1.0, END)  
 list1 = []  
 list2 = []  
 limitation = 0.0  
 word = txt.get()  
 word = word.lower()  
 word = re.sub(r'[^а-яА-Я0-9\s]', ' ', word)  
 length\_w = len(word)  
 trigram(length\_w, word, list1)  
 print(list1)  
 get\_data()  
 num\_lines, book = get\_data()  
 for i in range(0, num\_lines + 1):  
 line = book.readline().rstrip()  
 line = line.lower()  
 length = len(line)  
 trigram(length, line, list2)  
 union = set(list1).intersection(list2)  
 troll = 2 \* len(union)  
 if troll != 0:  
 p = (length\_w + length) / troll  
 if p <= 2:  
 limitation += 1.0  
 if limitation <= 5.0:  
 text.insert(limitation, "\n")  
 text.insert(limitation, line)  
 print(line, p)  
 list2.clear()  
 else:  
 list2.clear()  
 else:  
 list2.clear()  
  
  
lbl = Label(window, text="Введите слово")  
lbl.grid(column=0, row=0)  
lbl\_down = Label(window, text="Возможно вы имели ввиду")  
lbl\_down.grid(column=0, row=1)  
txt = Entry(window, width=25)  
txt.grid(column=1, row=0)  
txt.focus()  
text = Text(width=19, height=5, padx=0, pady=0)  
text.grid(column=1, row=1)  
btn = Button(window, text="Поиск", command=main)  
btn.grid(column=2, row=0)  
window.mainloop()