МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»
Кафедра Дискретной математики и информационных технологий

РАЗРАБОТКА ТЕКСТОВОГО РЕДАКТОРА С ИСПОЛЬЗОВАНИЕМ ЭЛЕМЕНТОВ НЕЧЕТКОЙ ЛОГИКИ

КУРСОВАЯ РАБОТА

студента 3 курса 321 группы
направления 09.03.01 — Информатика и вычислительная техника
факультета КНиИТ
Давиденко Алексея Алексеевича

Научный руководитель
Ассистент кафедры ДМиИТ _____ А.А. Трунов

Заведующий кафедрой
к. ф.-м.н., доцент _____ Л.Б. Тяпаев

Саратов 2020

СОДЕРЖАНИЕ

ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

HTML — HyperText Markup Language

CSS — Cascading Style Sheets

JS — JavaScript

ОПРЕДЕЛЕНИЯ

Metaphone — фонетический алгоритм для индексирования слов по их звучанию с учетом основных правил английского произношения;

Фреймворк — универсальная среда разработки программного обеспечения, которая обеспечивает особые функциональные возможности в рамках более большой программной платформы, облегчающая разработку решений;

Редакционное предписание — последовательность действий, необходимая для получения из первой строки второй кратчайшим образом;

ВВЕДЕНИЕ

Нечёткий поиск - это способ поиск информации, которая совпадает шаблону сравнения приблизительно или очень близкого шаблону значения. Алгоритмы нечёткого поиска применяются для распознавания текста, например, при занесении информации с отсканированных документов в базу, нахождения произошедших от некоторого слова слов, в поисковых системах, проверки орфографии и других областях [1].

Проблема нечеткого поиска текстовой информации может заключаться в следующем: имеется некоторый текст. Пользователь вводит в поле поиска запрос, представляющий из себя некоторое слово или последовательность слов, для которых необходимо найти в тексте все совпадения с запросом с учетом всех возможных допустимых различий. Например, при запросе "polynomial"нужно найти также слово "exponential".

Для оценки сходства двух слов в тексте используются специальные метрики нечеткого поиска, которые определяются как минимальное количество односимвольных операций (вставки, удаления, замены), необходимых для превращения одной строки в другую. В качестве метрик используются сходство Джардо-Винклера, расстояние Хемминга, расстояния Левенштейна и Дамерау-Левенштейна и другие.

Целью курсовой работы является реализация алгоритма нахождения расстояния Дамерау-Левенштейна для двух слов, поиск возможности минимизировать время поиска схожих слов и использование полученных алгоритмов в приложении.

Для достижения цели необходимо решить следующие задачи:

1. Реализовать алгоритмы нахождения схожести двух слов;
2. Создать словарь слов, по которому будет производиться поиск схожих слов;
3. Разработать приложение, текстовый редактор, использующий эти методы и словарь.

1 Алгоритмы поиска схожих слов

Для того, чтобы искать слова, в которых допущены ошибки, нужно выбрать метрику, которая будет удовлетворять следующим требованиям:

— Высокая скорость выполнения;

— Малые затраты памяти;

— Точное определение минимального расстояния между словами.

А так же, для того чтобы ускорить и сделать более точным поиск, необходимо производить поиск не только по самим словам, а по их фонетическим кодам. Для этого можно находить слова, расстояние между фонетическими кодами которых менее единицы.

1.1 Расстояние Левенштейна

Рассмотрим следующую задачу: имеется две строки s1 и s2, необходимо перевести либо s1 в s2, либо s2 в s1, используя операции:

— i: Вставка символа в произвольное место;

— d: Удаление символа с произвольной позиции;

— r: замена символа на другой.

Расстоянием Левенштейна для перевода s1 в s2 для рассматриваемой задачи будет d(s1, s2) - минимальное количество операций i/d/r для перевода s1 в s2, а редакционное предписание - перечисление операций для перевода с их параметрами.

Для расстояния Левенштейна справедливы следующие утверждения:

1. $d(s1, s2) \geq \|s1\| - \|s2\|$
2. $d(s1, s2) \leq \max(|s1|, |s2|)$
3. $d(s1, s2) = 0 \iff s1 = s2$

, где |s| - это длина строки s.

Искомое расстояние формируется через вспомогательную функцию D(m, n), находящую редакционное расстояние для срезов s1[0, m] и s2[0, n], где D(i, j) находится по формуле 1

$$D(i,j) = \begin{cases} 0 & ; i = 0, j = 0 \\ i & ; j = 0, i > 0 \\ j & ; i = 0, j > 0 \\ D(i-1, j-1) & ; s1[i] = s2[j] \\ \min \begin{pmatrix} D(i, j-1) \\ D(i-1, j) \\ D(i-1, j-1) \end{pmatrix} + 1 & : j > 0, i > 0, s1[i] \neq s2[j] \end{cases} \quad (1)$$

, где (i, j) - клетка матрицы, в которой мы находимся на данном шаге. [2]

Здесь $D(i, 0) =$ и $D(0, j) = j$ получены из соображения, что любая строка может получиться из пустой, добавлением нужного количества нужных символов, любые другие операции будут только увеличивать оценку.

В общем случае имеем D(i, j) по формуле 2 [2]

$$D(i,j) = D(i-1, j-1), s1[i] = s2[j], \quad (2)$$

иначе по формуле 3

$$D(i,j) = \min \begin{pmatrix} D(i, j-1) \\ D(i-1, j) \\ D(i-1, j-1) \end{pmatrix} + 1 \quad (3)$$

.

В данном случае мы выбираем наиболее выгодную операцию: удаление символа (D(i - 1, j)), добавление (D(i, j - 1)) или замена (D(i - 1, j - 1)).

Вычисление матрицы D можно произвести, используя следующий псевдокод:

```
1 input: strings a[1..length(a)], b[1..length(b)]
2 output: distance, integer
3
4 let d[0..length(a), 0..length(b)] be a 2-d array of integers, dimensions
     length(a)+1, length(b)+1
5 // note that d is zero-indexed, while a and b are one-indexed.
6
7 for i := 0 to length(a) inclusive do
```

```
    d[i, 0] := i
for j := 0 to length(b) inclusive do
    d[0, j] := j

for i := 1 to length(a) inclusive do
    for j := 1 to length(b) inclusive do
        if a[i] = b[j] then
            cost := 0
        else
            cost := 1
        d[i, j] := minimum(d[i-1, j] + 1,      // deletion
                           d[i, j-1] + 1,      // insertion
                           d[i-1, j-1] + cost)  // substitution
return d[length(a), length(b)]
```

Результат работы алгоритма для слов “Пять” и “Семь” представлен в таблице 1.1.

Таблица 1.1 – Пример работы алгоритма

		П	Я	Т	Ь
	0	1	2	3	4
С	1	1	2	3	4
Е	2	2	2	3	4
М	3	3	3	3	4
Ь	4	4	4	4	3

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна - к операциям замены, удаления и вставки добавляется операция перестановки символов.

Большинство опечаток могут быть получены из правильного написания по нескольким простым правилам. Дамерау указывает, что 80 процентов всех орфографических ошибок являются результатом [3]:

— Перестановки двух букв;
— Добавления буквы;
— Удаления буквы;
— Написания неправильной буквы.

Для нахождения расстояния Дамерау-Левенштейна используется два алгоритма: упрощенный алгоритм, в котором предполагается, что любой срез

может быть редактирован не более одного раза, и корректный алгоритм, в котором этого ограничения нет.

В обоих алгоритмах редакционное предписание имеет вид:

$$D(i,j) = \min \begin{cases} 0 & ; i = 0, j = 0 \\ D(i-1,j) + 1 & ; i > 0 \\ D(i,j-1) + 1 & ; j > 0 \\ D(i-1;j-1) + 1_{s1[i] \neq s2[j]} & ; i > 0, j > 0 \\ D(i-2,j-2) + 1 & ; i > 1, j > 1, a[i] = b[j-1], \\ & a[i-1] = b[j] \end{cases}$$

Различие между алгоритмом нахождения расстояния Левенштейна и упрощенным алгоритмом нахождения расстояния Дамерау-Левенштейна состоит в добавлении условия перестановки в последнем. Псевдокод алгоритма:

```
1  input: strings a[1..length(a)], b[1..length(b)]
2  output: distance, integer
3
4  let d[0..length(a), 0..length(b)] be a 2-d array of integers, dimensions
       length(a)+1, length(b)+1
5  // note that d is zero-indexed, while a and b are one-indexed.
6
7  for i := 0 to length(a) inclusive do
8      d[i, 0] := i
9  for j := 0 to length(b) inclusive do
10     d[0, j] := j
11
12 for i := 1 to length(a) inclusive do
13     for j := 1 to length(b) inclusive do
14         if a[i] = b[j] then
15             cost := 0
16         else
17             cost := 1
18         d[i, j] := minimum(d[i-1, j] + 1,      // deletion
19                            d[i, j-1] + 1,      // insertion
20                            d[i-1, j-1] + cost)  // substitution
21         if i > 1 and j > 1 and a[i] = b[j-1] and a[i-1] = b[j] then
22             d[i, j] := minimum(d[i, j],
23                                d[i-2, j-2] + 1)  // transposition
24 return d[length(a), length(b)]
```

Корректный алгоритм находит расстояние без ограничения возможных

перестановок в подстроках, из-за этого ему необходим дополнительный параметр - размер алфавита Σ, такой, что все буквы данных строк будут находиться в массиве $[0, |\Sigma|]$ [4]. Псевдокод алгоритма:

```
input: strings a[1..length(a)], b[1..length(b)]
output: distance, integer

da := new array of |Sigma| integers
for i := 1 to |Sigma| inclusive do
    da[i] := 0

let d[-1..length(a), -1..length(b)] be a 2-d array of integers, dimensions
    length(a)+2, length(b)+2
// note that d has indices starting at -1, while a, b and da are one-indexed.

maxdist := length(a) + length(b)
d[-1, -1] := maxdist
for i := 0 to length(a) inclusive do
    d[i, -1] := maxdist
    d[i, 0] := i
for j := 0 to length(b) inclusive do
    d[-1, j] := maxdist
    d[0, j] := j

for i := 1 to length(a) inclusive do
    db := 0
    for j := 1 to length(b) inclusive do
        k := da[b[j]]
        l := db
        if a[i] = b[j] then
            cost := 0
            db := j
        else
            cost := 1
        d[i, j] := minimum(d[i-1, j-1] + cost,  //substitution
                           d[i,   j-1] + 1,     //insertion
                           d[i-1, j  ] + 1,     //deletion
                           d[k-1, l-1] + (i-k-1) + 1 + (j-l-1)) //
    transposition
da[a[i]] := i
return d[length(a), length(b)]
```

Здесь da представляет собой массив, хранящий в себе индексы последних совпадений s1[*] с s2[j] (i‘ $<$ i, j: s1[i‘] $=$ s2[j]), db - индексы последних совпадений s2[*] с s1[i] (i, j’ $<$ j: s2[j’] $=$ s1[i]) .

1.3 Metaphone

Для того, чтобы уменьшить время поиска похожих слов, можно предварительно воспользоваться одним из фонетических алгоритмов.

Фонетические алгоритмы сопоставляют словам с похожим произношением одинаковые коды, что позволяет производить поиск таких слов на основе их фонетического сходства [5].

Например, слова ‘Desert’ и ‘Dessert’ будут иметь схожие фонетические коды.

Одним из таких алгоритмов является Metaphone. Этот алгоритм преобразует слова к кодам переменной длины, состоящим только из букв, по сложным правилам. Алгоритм включает в себя следующие шаги [6]:

1. Удаление повторяющихся соседних букв кроме буквы C;
2. Если слово начинается с ‘KN’, ‘GN’, ‘PN’, ‘AE’, ‘WR’, убирается первая буква (‘KN’ -> ‘N’, ‘GN’ -> ‘N’, ...);
3. Опускается ‘B’ в ‘MB’, если ‘MB’ - суффикс;
4. ‘C’ преобразуется в ‘X’, если за ним следует ‘IA’ или ‘H’ (только если он не является частью ‘-SCH-’, в этом случае он преобразуется в ‘K’). ‘C’ преобразуется в ‘S’, если за ним следуют ‘I’, ‘E’ или ‘Y’. В остальных случаях ‘C’ преобразуется в ‘K’;
5. ‘D’ преобразуется в ‘J’, если за ним следуют ‘GE’, ‘GY’ или ‘GI’. В противном случае ‘D’ преобразуется в ‘T’;
6. Удаляется ‘G’, если за ним следует ‘H’, причем ‘H’ стоит не в конце слова и не перед гласным. Также удаляется ‘G’, если за ним следует ‘N’ или ‘NED’, и он является окончанием;
7. ‘G’ преобразуется в ‘J’, если до ‘I’, ‘E’ или ‘Y’, и это не в ‘GG’. В противном случае ‘G’ преобразуется в ‘K’;
8. Опускается ‘H’, если он стоит после гласного и не перед гласным;
9. ‘CK’ преобразуется в ‘K’;
10. ‘PH’ преобразуется в ‘F’;
11. ‘Q’ преобразуется в ‘K’;
12. ‘S’ преобразуется в ‘X’, если за ним следуют ‘H’, ‘IO’ или ‘IA’;
13. ‘T’ преобразуется в ‘X’, если за ним следует ‘IA’ или ‘IO’. ‘TH’ преобразуется в ‘0’. ‘T’ опускается, если за ним следует ‘CH’;
14. ‘V’ преобразуется в ‘F’;

15. ‘WH’ преобразуется в ‘W’, если он стоит в начале. ‘W’ опускается, если за ним не следует гласная;
16. «X» преобразуется в «S», если он стоит в начале. В противном случае «X» преобразуется в «KS»;
17. ‘Y’ опускается, если за ним не следует гласная;
18. ‘Z’ преобразуется в ‘S’;
19. Опускаются все гласные, кроме начального.

В последствие алгоритм Metaphone был улучшен, была выпущена вторая версия алгоритма,которая получила название Double Metaphone, в которой, в отличие от первой версии, применимой только к английскому языку, учитывалось происхождение слов, особенности их произношения [6]. Для таких слов результатом работы являются два кода - основной вариант произношения и альтернативный [5]. Алгоритм Double Metaphone сложнее, чем его предшественника, увидеть его можно в статье Лоуренса Филипса ‘The double metaphone search algorithm’ `https://dl.acm.org/doi/10.5555/349124.349132`.

Пример работы алгоритма: ‘My String’ будет преобразовано к ‘MSTRNK’.

Алгоритм Metaphone был адаптирован к русскому языку. Для русского языка алгоритм состоит из пяти шагов [6]:

1. Преобразование гласных путем следующих подстановок: О, Ы, Я -> А; Ю -> У; Е, Ё, Э, ЙО, ЙЕ -> И;
2. Оглушение согласный букв, за которыми следует любая согласная, кроме Л, М, Н или Р, либо согласных на конце слова путем следующих подстановок: Б -> П; З -> С; Д -> Т; В -> Ф; Г -> К;
3. Удаление повторяющихся букв;
4. Преобразование суффикса слова путем следующих подстановок: УК, ЮК -> 0; ИНА -> 1; ИК, ЕК -> 2; НКО -> 3; ОВ, ЕВ, ИЕВ, ЕЕВ -> 4; ЫХ, ИХ -> 5; АЯ -> 6; ЫЙ, ИЙ -> 7; ИН -> 8; ОВА, ЕВА, ИЕВА, ЕЕВА -> 9; ОВСКИЙ -> @; ЕВСКИЙ -> #; ОВСКАЯ -> $; ЕВСКАЯ -> %;
5. Удаление букв Ъ, Ь и дефиса.

Из-за небольшого числа правил, адаптированный для русского языка алгоритм Metaphone не отождествляет некоторые схожие фонетически слова.

2 Средства для программной реализации

Для написания программы, которая бы реализовала алгоритмы, описанные выше, нужно, в первую очередь, выбрать подход, которого мы будем придерживаться при разработке. Есть два подхода: разработка нативных приложений, т.е. приложений, которые работают только на определённой платформе или на определённом устройстве, и разработка кроссплатформенных приложений, т.е. приложений, которые способны работать с двумя и более платформами.

В следствие чего был выбран кроссплатформенный подход, т.к. он позволяет разрабатывать программу на одной операционной системе и знать, что разработанная программа будет работать и на других ОС, необходимо лишь перекомпилировать приложение для нужной платформы.

Выбирал я между двумя фреймворками языков, Qt (C++) и Electron (JS), с помощью которых можно писать кроссплатформенные приложения, и, для того чтобы определиться, какой фреймворк использовать, необходимо рассмотреть несколько критериев:

— Лёгкость поддержки приложения;
— Легкость разработки приложения;
— Распространённость фреймворка.

В итоге Electron, оказался наиболее подходящим для разработки и поддержания.

2.1 Electron

Electron - это фреймворк для разработки настольных кроссплатформенных приложений с использованием HTML, CSS и JS [7]. Его особенность состоит в том, что если ты знаешь как разрабатывать сайты, например, то ты сможешь разработать и настольное приложение. По сути, приложение, написанное на Electron представляет собой окно браузера, в котором открыто единственное окно — ваше приложение.

Процесс разработки на Electron разбит на две взаимно зависимые части: разработка интерфейса приложения (фронтэнда) и разработку логической части приложения (бекэнда). Обе эти части можно написать используя лишь JS, HTML и CSS.

2.2 HTML

HTML - это стандартизированный язык гипертекстовой разметки документов во Всемирной паутине. Браузер может интерпретировать описанный с помощью HTML документ и отобразить его структуру на экране пользователя.

После загрузки веб-страницы, браузер создаёт DOM - Document Object Model - объектную модель документа этой страницы. Благодаря этой модели, содержимое сайта можно прочитать и изменить с помощью скриптов, в частности, с помощью JS. Благодаря этому можно описать данные в виде набора утверждений и формул, изменение которых ведет к автоматическому перерасчёту всех зависимостей, сделать сайт реактивным с помощью, например, JS.

Во многих фронтэнд-фреймворках реализована реактивность. Я выбрал VueJS как один из самых популярных и прогрессивных фреймворков.

2.3 Vue

VueJS позволяет декларативно отображать данные в DOM с помощью простых шаблонов. Например, следующий пример кода создаст в DOM компонент, содержащий приветствие:

```
1  <div id = "app">
2    {{message}}
3  </div>
4
5  var app = new Vue ({
6    el: "#app",
7    data: {
8      message: "Hello"
9    }
10 })
```

Данные и DOM теперь реактивно связаны - при изменении данных, DOM автоматически перестроится.

В Vue каждое поле данных автоматически разбивается на пары геттер и сеттер. С их помощью Vue может следить, какие данные читались или изменялись и может определить, какие факторы влияют на отрисовку отображения.

Каждому экземпляру компонента приставлен связанный с ним экземпляр наблюдателя, который помечает все поля, затронутые при отрисовке,

как зависимые. Когда вызывается сеттер поля, помеченного как зависимость, этот сеттер уведомляет наблюдателя, который, в свою очередь, инициирует повторную отрисовку компонента [8].

С помощью этого можно разрабатывать крупные проекты, не отвлекаясь на проблему синхронизации данных.

3 Практическая часть

Практическая часть

ЗАКЛЮЧЕНИЕ

Вывод

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Мосалев П.М. Обзор методов нечеткого поиска текстовой информации [текст] // Вестник МГУП. 2013. №2. URL: https://cyberleninka.ru/article/n/obzor-metodov-nechetkogo-poiska-tekstovoy-informatsii (дата обращения: 06.10.2019). Загл. с экрана Яз. рус

2 Habr [Электронный ресурс] // https://habr.com/ – Вычисление редакционного расстояния URL: https://habr.com/ru/post/117063/ (дата обращения 06.10.2019). Загл. с экрана Яз. рус

3 James L. Peterson Computer Programs for Detecting and Correcting Spelling Errors [текст] / James L. Peterson // Communications of the ACM. 1980. №23. С. 676-687. URL: https://dl.acm.org/doi/10.1145/359038.359041 (дата обращения: 07.10.2019). Загл. с экрана Яз. англ

4 Leonid Boytsov ndexing Methods for Approximate Dictionary Searching: Comparative Analysis [текст] / Leonid Boytsov // J. Exp. Algorithmics. 2011. №16. С. 1-93. URL: https://doi.org/10.1145/1963190.1963191 (дата обращения: 10.11.2019). Загл. с экрана Яз. англ

5 Habr [Электронный ресурс] // https://habr.com/ – Фонетические алгоритмы URL: https://habr.com/ru/post/114947/ (дата обращения 11.11.2019). Загл. с экрана Яз. рус

6 Выхованец Валерий Святославович, Ду Цзяньмин, Сакулин Сергей Александрович Обзор алгоритмов фонетического кодирования [текст] /Выхованец Валерий Святославович, Ду Цзяньмин, Сакулин Сергей Александрович // УБС. 2018. №73. URL: https://cyberleninka.ru/article/n/obzor-algoritmov-foneticheskogo-kodirovaniya (дата обращения: 11.11.2019) Загл. с экрана Яз. рус

7 Habr [Электронный ресурс] // https://habr.com/ – Electron: разработка настольных приложений с использованием HTML, CSS и JavaScript URL: https://habr.com/ru/company/ruvds/blog/436466/ (дата обращения 05.12.2019) Загл. с экрана Яз. рус

8 vue.js [Электронный ресурс] // https://ru.vuejs.org/ – Подробно о реактивности URL: https://ru.vuejs.org/v2/guide/reactivity.html (дата обращения 12.12.2019) Загл. с экрана Яз. рус

ПРИЛОЖЕНИЕ А

Приложение А