Санкт-Петербургский государственный университет

Математико-механический факультет

Кафедра системного программирования

Реализация нечеткого поиска в условиях низкого порога схожести в системе обработки документов

Курсовая работа студента 445 группы

Чередника Кирилла Евгеньевича

Научные руководители:

Чернышев Г.А.,

Ассистент кафедры информатики СПбГУ

Смирнов К.К.,

Инженер ЗАО “Ланит-Терком”

Санкт-Петербург

2013

Оглавление

Введение

Задача нечеткого поиска формулируется как нахождение всех строк из данного набора, близких к заданной строке по заданной метрике на пространстве строк.

Нечеткий поиск является важной подзадачей во многих современных системах. Он применяется в таких областях как информационный поиск, биоинформатика, проверка орфографии, и т. д. В информационном поиске он полезен при исправлении опечаток в поисковых запросах. То есть, если пользователь ошибся в написании запроса или не знал правильного написания термина, то поисковая система предложит правильный вариант. Это возможно благодаря тому, что поисковая система может, ориентируясь на словарь, понять, что введенное слово неправильно, и предложить наиболее близкое к нему (если расстояние между словами не очень велико), среди тех, что есть в словаре. Подобные алгоритмы применяются в таких поисковых системах как Google, Yandex и др. В биоинформатике нечеткий поиск применяется, например, для различных операций с последовательностями ДНК [1].

На данный момент существует множество подходов к решению задачи нечеткого поиска. Разработаны метрики для сравнения строк, алгоритмы, способные быстро вычислять расстояние в заданной метрике или проверять соответствие двух слов друг другу по заданной метрике и допустимому отклонению, а также множество структур данных и алгоритмов оперирования множествами строк для реализации нечеткого поиск. Среди них можно отметить такие структуры данных как Trie [2], FQA (Fixed Queries Array) [3], различные метрические деревья – M-tree [4], BK-tree [5], VP-tree[6].

Несмотря на то, что предметная область, связанная с нечетким поиском, достаточно стара, она до сих пор актуальна. Активно публикуются статьи, проводятся различные соревнования прототипов систем, включающих в себя нечеткий поиск. Например, EDBT 2013 String Similarity Search/Join Competition, SIGMOD 2013 Programming Contest. В них ставится определенная задача и дается время на изучение алгоритмов и реализацию прототипа (около нескольких месяцев).

Данная курсовая работа выполнялась в рамках одного из таких соревнований, а именно: ACM SIGMOD Programming Contest 2013. Полный состав команды ‘RotaFortunae’ (‘Колесо Фортуны’, лат.): Федотовский П.В., Ерохин Г.А., Чередник К.Е.

Общий обзор системы

На вход системе поступает поток документов и запросов. Цель – для каждого документа найти все запросы из числа активных, которые ему удовлетворяют. В качестве примера можно рассмотреть твиттер. Документы в данном случае это твиты, которые публикуются пользователями. Запросы это хэштеги, на которые пользователь хочет подписаться. Хэштег – это обычное слово, если оно встречается в твите, то все пользователи, которые на него были подписаны, увидят этот твит в своей новостной ленте. Другим примером могут быть новостные рассылки. Пользователи указывают ключевые слова, и если новый документ их содержит, то его увидят все заинтересованные пользователи.

Более формально, нужно определить понятия документа и запроса. Документ – это множество слов, ограничения на их количество нет. Однако в задаче контеста длина документов ограничена сверху – максимальное количество символов 1.000.000. Запрос – небольшой набор слов, используемая метрика и максимальное допустимое отклонение.

Запрос считается подходящим документу, если для каждого слова из запроса в документе найдется слово, расстояние до которого от слова в запросе не превосходит максимального отклонения по данной метрике.

Общая постановка задачи следующая: по каждому документу определить все запросы из числа активных, которые ему подходят. Основной особенностью системы является наличие нечеткого поиска при поиске соответствия.

Детальное описание системы дано в Приложении 1.

Постановка задачи

Целью данной курсовой работы является разработка методов в описанной системе, которые реализуют нечеткий поиск.

Для достижения этой цели были поставлены следующие задачи:

- Обзор и анализ алгоритмов и структур данных
- Разработка с учетом специфики системы.
  - малый порог разницы (<=3)
  - короткие слова (<=31)
  - современное оборудование

Обзор существующих решений

Учитывая то, что данная задача является довольно популярной, существует множество исследований и разработок в данной области. Но многие из этих решений опираются на некоторые специфики конкретных задач, что делает их малопригодными в данных условиях. К примеру, в задачах биоинформатики далеко не всегда нужен точный результат, поэтому в подобной ситуации часто используются приближенные методы. Или в задачах информационного поиска зачастую можно потратить довольно много ресурсов на обработку “документов”, так как они по большей части задач довольно статичные. Таким образом, у нас возникают дополнительные трудности в определении эффективности конкретного алгоритма для нашей задачи и адаптации этого алгоритма.

Мы будем изучать два аспекта нечеткого поиска: это фильтрация (построение дополнительных структур данных для быстрого отфильровывания заведомо далеких друг от друга строк в заданной метрике) и сравнение (проверка для двух конкретных строк на то, что расстояние между ними в заданной метрике, не превосходит некоторого небольшого порогового значения).

Фильтрация

Теперь перейдем к конкретным подходам в реализации предварительной фильтрации.

Разобьем методы фильтрации по метрикам в терминах, которых будет производиться поиск.

- Дискретная метрика:

Хэш-таблицы. Одним из самых эффективных способов точного поиска в данной ситуации будет построение хэш-таблиц. В таком случае, мы за O(1) можем сразу определить есть ли такое слово во множестве. Так как время поиска при использовании такого алгоритма пренебрежительно мало по сравнению с поиском по остальным метрикам, а реализация не очень требовательна к памяти, то другие алгоритмы для данной метрики можно не рассматривать.

- Метрики Хэмминга и Левенштейна:

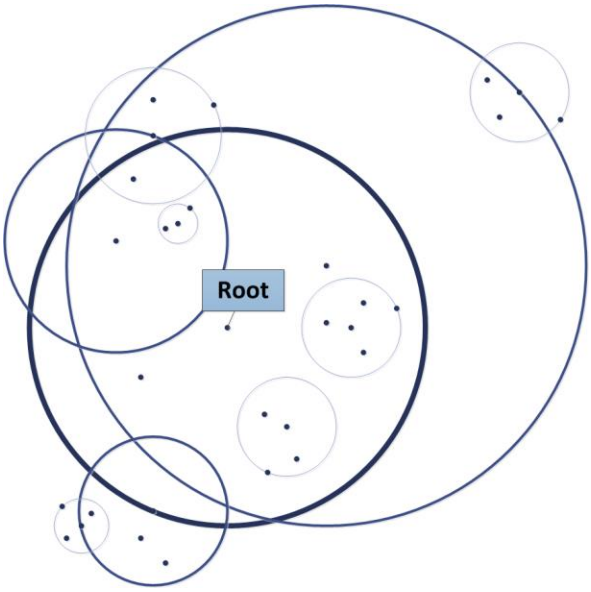
M-tree. Древовидная структура данных, строящаяся на элементах метрического пространства. Каждая вершина дерева содержит элемент пространства, который уникально идентифицирует данную вершину, список дочерних узлов, радиус данной вершины, где для любой вершины расстояние от неё до любого из её потомков не превосходит радиуса (под расстоянием между вершинами понимается расстояние между соответствующими элементами метрического пространства).

Как легко видеть, при качественном построении M-tree позволяет довольно эффективно реализовывать поиск всех элементов множества, для которых расстояние до конкретного объекта не превосходит некоторого порогового значения.

Но есть несколько проблем с использованием данной структуры в этой задаче:

- Сложность реализации эффективной многопоточной вставки элементов: например, можно реализовать M-tree с использованием GiST, но это довольно трудоемкая задача.

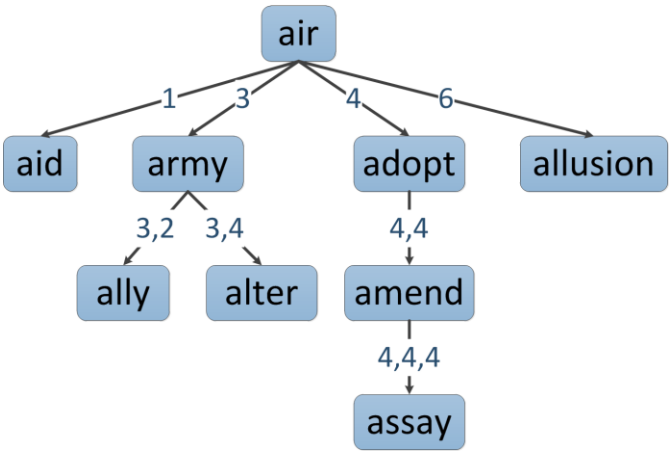
- Проблемы в реализации, эффективно использующей кэш: было бы эффективнее хранить вершины, принадлежащие одному узлу близко в памяти, но это требует дополнительных расходов на перераспределение элементов.
- Необходимость многократного вызова связанных с метрикой функций при построении: при вставке новых элементов в дерево необходимо много раз вычислять расстояние и проверять, не превосходит ли расстояние между двумя словами некоторого порогового значения, где пороговое значение может быть довольно большим. А если использовать это дерево для пространств с “дорогой” метрикой, то это может сильно увеличить время построения структуры. К “дорогим” метрикам можно, в частности, отнести метрику Левенштейна, так как, как будет показано далее, время работы известных алгоритмов для подобных вычислений и проверок, по крайней мере линейно зависит от величины порогового значения (если считать, что пороговое значение не превосходит длины большего из двух слов).
- Неэффективность при работе с короткими словами и большим пороговым значением: при использовании этой структуры для индексирование пространства с метрикой Левенштейна, если длины слов невелики, а пороговое значение высоко (подобная ситуация довольно часта в данной задаче), то накладные расходы, связанные с поиском по M-tree и его построением, становятся слишком большими, учитывая то, что фильтрация может отсеять только малую часть результатов.



BK-tree. Древовидная структура данных, строящаяся на элементах метрического пространства. Каждая вершина дерева содержит элемент метрического пространства, который уникально идентифицирует данную вершину, список дочерних узлов, где вставка элементов происходит следующим образом: если дерево пусто, то ставим корнем новый элемент, иначе рекурсивно находим расстояние R между данным словом и корнем текущего поддерева и вставляем этот элемент в поддерево с индексом R.

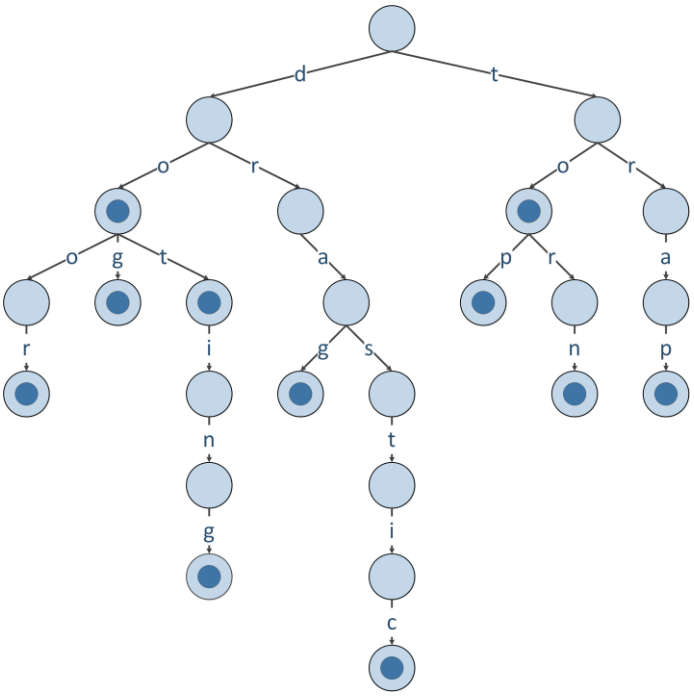
Проблемы с использованием этой структуры данных во многом сходны с проблемами связанными с M-tree:

- Проблемы в реализации, эффективно использующей кэш.
- Неэффективность при работе с короткими словами и большим пороговым значением.

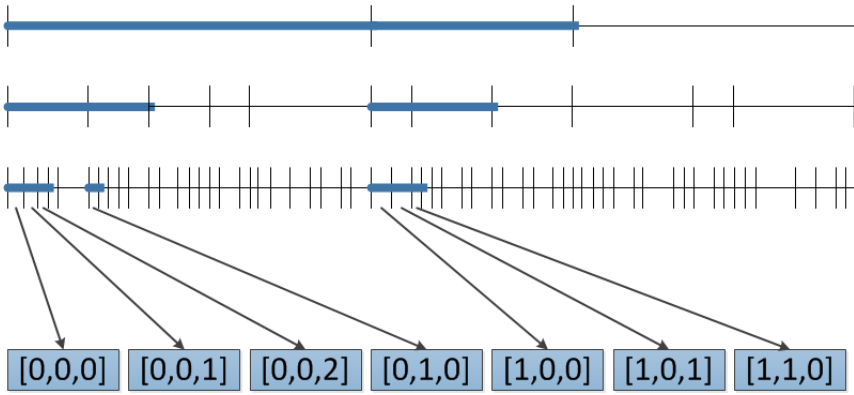


Trie. Древовидная структура данных, в которой каждый узел (кроме корневого), хранит некоторый символ. Строится она следующим образом: при вставке нового слова, оно разбивается на символы, в корне выбирается дочерний элемент, соответствующий первому символу, у выбранного поддерева выбирается дочерний элемент, соответствующий второму символу и т.д. В полученной в конце вершине отмечается, что в ней оканчивается некое слово. Тогда можно найти все слова из множества близкие к данному, совершая обход дерева в глубину, накапливая ошибку, пока та не превысит

порогового значения.



FQA. Структура данных, строящаяся следующим образом: из набора выбирается несколько опорных элементов, далее для каждого элемента набора считается вектор расстояний до опорных элементов. Все элементы набора сортируются лексикографически по векторам расстояний. После этого поиск близких элементов к данному происходит следующим образом. Считаем расстояние от искомого элемента до опорных $[q_0, \ldots, q_n]$ и сравниваем этот элемент только с теми, которые отстоят от данного не более чем на пороговое значение в определенной метрике пространства векторов. А все подходящие элементы ищутся при помощи двоичного поиска.



VP-tree. Двоичное дерево, строящееся следующим образом: если элементов во множестве больше некоторого порогового значения (связанного со структурой дерева), то из этого множества выбирается опорный элемент. Далее все элементы разбиваются на два, примерно равных по размеру, множества, где расстояние от любого элемента первого множества до опорного элемента меньше расстояния от любого элемента второго множества до опорного элемента.

Поиск же осуществляется следующим образом: в зависимости от того, превосходит ли T + D M, где T - пороговое значение (связанное с искомым словом), D – расстояние от искомого слова до опорного значения текущей вершины, M – минимальное значение расстояний от элементов второго поддерева до опорного элемента, мы рекурсивно проверяем либо только первое дерево, либо и первое, и второе.

Здесь опять же возникают уже известные нам проблемы:

- Сложность реализации эффективной многопоточной вставки элементов.
- Проблемы в реализации, эффективно использующей кэш.
- Необходимость многократного вызова функций связанных с метрикой при построении.
- Неэффективность при работе с короткими словами и большим пороговым значением.

Подсчёт расстояния

Теперь перейдем к существующим решениям в области подсчёта расстояния между двумя словами и определении, не превышает ли расстояние между двумя словами порогового значения, в терминах соответствующей метрики.

- Дискретная метрика

На этапе фильтрации мы выявили, есть ли данное слово во множестве, поэтому рассматривать методы точного сравнение двух слов мы не будем.

- Метрика Хэмминга

Расстояние Хэмминга между двумя словами подсчитывается с помощью прохода по строкам и подсчёта количества несовпадающих символов на соответствующих позициях. А для проверки на то, что расстояние не превосходит порогового значения можно сделать ранний выход из цикла при достижении количества несовпадающих символов этого значения. А уже этот проход по строкам можно оптимизировать, например, разворачиванием цикла.

- Метрика Левенштейна.

Прямой подход

Расстояние между двумя строками a и b может быть вычислено как $\mathrm{lev}_{a,b}(|a|, |b|)$, где

$$\mathrm{lev}_{a,b}(i,j) = \begin{cases} \max(i,j) & \text{if } \min(i,j) = 0, \\ \min \begin{cases} \mathrm{lev}_{a,b}(i-1,j) + 1 \\ \mathrm{lev}_{a,b}(i,j-1) + 1 \\ \mathrm{lev}_{a,b}(i-1,j-1) + [a_i \neq b_j] \end{cases} & \text{otherwise.} \end{cases}$$

При таком подходе для вычисления расстояния необходимо вычислить матрицу значений $\mathrm{lev}_{a,b}(i, j)$, для $i \in [0, |a|]$, $j \in [0, |b|]$. Что можно сделать за $O(n^2)$.

Отсечение Укконена

Для проверки того, что расстояния Левенштейна не превосходит некоторого порогового значения K, то можно использовать таблицу для вычисления расстояния, но не обязательно вычислять значение $lev_{a,b}(i, j)$, для всех i, j, а можно их вычислить только для $|i - j| \leq K$. Данная оптимизация позволяет проверять два слова на близость по метрике Левенштейна за $O(Kn)$.

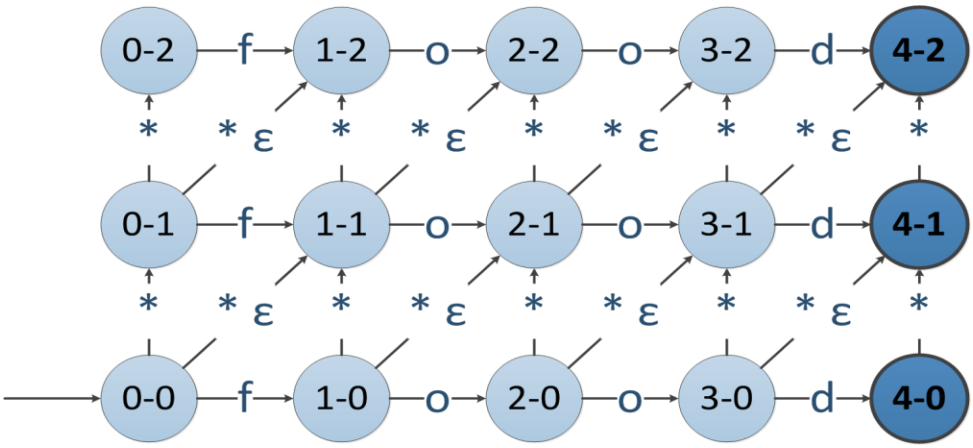
		E	L	E	P	H	A	N	T
	0	1	2	3	4	5	6	7	8
R	1	1	2	3	4	5	6	7	8
E	2	1	2	2	3	4	5	6	7
L	3	2	1	2	3	4	5	6	7
E	4	3	2	1	2	3	4	5	6
V	5	4	3	2	2	3	4	5	6
A	6	5	4	3	3	3	3	4	5
N	7	6	5	4	4	4	4	3	4
T	8	7	6	5	5	5	5	4	3

Автоматы

Для каждого слова и соответствующего расстяния мы можем построить недетерменированный автомат, где состояния будем обозначать как “n-k”, где n - количество уже обработанных символов, k - количество ошибок. а переходы строятся по такому принципу: горизонтальные переходы означают что модификаций не было, вериткальные означают вставки, диагональные переходы означают замены и удаления. Конечными же состояниями здесь будут являться те состояние, для которых n = длине исходного слова.

Подобные автоматы позволяют проверять расстояние Левенштейна за $O(k^2n)$.

Также стоит отметить, что подобные недетерминированным автоматы можно преобразовать в детерминированные. Но будучи приведенными к детерминированным, автоматы, зачастую, начинают занимать неприемлемо много памяти, а именно: число узлов автомата растет экспоненциально[??].


0-2
f
1-2
o
2-2
o
3-2
d
4-2
0-1
f
1-1
o
2-1
o
3-1
d
4-1
0-0
f
1-0
o
2-0
o
3-0
d
4-0
ε

Предложенные решения

По результатам обзора было решено использовать следующие структуры данных и алгоритмы.

Фильтрация

Для фильтрации слов в пространстве с метриками Хэмминга и Левенштейна, учитывая специфику задачи, были предложены следующие подходы.

- Множество слов с метрикой **Хэмминга** и пороговым расстоянием **N**.

Хэш-таблицы

Для всех возможных длин слов строим **N + 1** хэш-таблицу (получается всего **29 * (N + 1)** таблиц для каждого **N**) следующим образом: разбиваем слово на **N + 1** непересекающихся подстрок, где длины первой части подстрок превосходят длины второй части подстрок максимум на **1** (например для **N = 3** строка "**abcdefg**" разбивается на подстроки "**ab**", "**cd**", "**ef**", "**g**"). для каждой подстроки **S** кладем в соответствующую хэш-таблицу пару **"S -> слово"**.

Тогда для поиска слов, близких к данному, необходимо разбить это слово на подстроки тем же образом и просмотреть соответствующие элементы хэш-таблиц и проверить полученные результаты (проверка не обязательна для **N = 3** и длины слова **4**). Для избегания многократной проверки одного слова, полученного из разных хэш-таблиц, можно использовать time-stamps.

- Множество слов с метрикой **Левенштейна** и пороговым расстоянием **N**.

Хэш-таблицы (1)

Для всех возможных длин слов строим одну хэш-таблицу (получается всего **29** таблиц для каждого **N**) следующим образом: Пусть длина слова - **L**. Тогда для каждого **L'** из **[max(L - N, 4), min(L + N, 32)]** выделяем из слова все **L' / (N + 1)**-граммы ("/" - целочисленное деление) и кладем в хэш-таблицу для длины **L'** пары **"q-грамма -> слово"**.

Тогда для поиска слов, близких к данному, необходимо разбить это слово на N + 1 непересекающихся подстроки одинаковой максимальной длины и для каждой этой подстроки просмотреть соответствующие элементы хэш-таблицы (которая определяется длиной слова) и проверить полученные результаты (плохо работает для коротких слов и большого расстояния Левенштейна, т.к. q-граммы будут односимвольными).

Хэш-таблицы (2)

Для всех возможных длин слов строим **N + 1** хэш-таблицу (получается всего **29 * (N + 1)** таблиц для каждого **N**) следующим образом: Пусть длина слова - **L**. Тогда для каждого **L'** из **[max(L - N, 4), min(L + N, 32)]** выделяем из слова **K**-граммы, где **K = L' / (N + 1)** ("/" - целочисленное деление) с позициями из диапазона **[max(K * X - N, 0), min(K * X + N, L' - K)]**, где **X** из **[0, N]** и кладем в одну из хэш-таблиц (в зависимости от **X**) для длины **L'** пар **"K-грамма -> слово"**.

Тогда для поиска слов, близких к данному, необходимо разбить это слово на **N + 1** непересекающихся подстроки одинаковой максимальной длины и для каждой этой подстроки просмотреть соответствующие элементы хэш-таблиц (которые определяются длиной слова и номером подстроки) и проверить полученные результаты (плохо работает для коротких слов и большого расстояния Левенштейна, т.к. q-граммы будут одно символьными). Для избегания многократной проверки одного слова, полученного из разных хэш-таблиц, можно использовать time-stamps.

Битовые маски

Так как размер алфавита небольшой (26 символов), то мы можем для каждого слова посчитать массив битовых масок длинной равной L, где *L* - длина слова + N, который будет содержать в себе следующие значения: на позиции *i* маска состоит из нулей кроме тех позиций *p*, для которых $\exists\, j \in [\max(i - N, 0), \min(i + N, L)]$, что код символа в позиции *j* равен *p*.

Теперь при проверке близости двух слов одинаковой длины, для которых известны маски, можно проверить количество ненулевых элементов среди побитовых & масок с индексами в интервале $[0, \max(L_a, L_b)]$ этих слов и если оно превышает N, то можно сазу сказать, что и расстояние между словами превышает N.

Подсчёт расстояния

Для определения того, не превышает ли расстояние между двумя словами порогового значения, в метриках Левенштейна или Хэмминга, нами были предложены следующие решения.

- Метрика Хэмминга.

SSE4.2

В реализации был использован 128 битный тип данных __m128i, предоставляемый компилятором gcc. Современные процессоры, поддерживающие расширение SSE4.2, могут обрабатывать этот тип данных наиболее эффективным образом: за две инструкции процессора можно вычислить расстояние Хэмминга между двумя строками длиной 16 байт.

_mm_cmpestrm(__m128i a, int la, __m128i b, int lb, const int mode) // вычислить маску

_mm_popcnt_u64(unsigned __int64 a) // подсчитать число единиц

- Метрика Левенштейна.

Автоматы

Так как пороговое расстояние не больше 3 (что не очень много), то мы можем разбить сравнения на 9 типов по парам **(***пороговое значение***,** *разница длин строк***)**, так как расстояние Левенштейна не меньше разницы длин сравниваемых строк.

Каждый тип автомата строится, основываясь на анализе возможных операций для преобразования одной строки в другую, количество которых не превышает порогового значения. Например, при пороговым значении равным 3 и разнице длин равной 3, состояниями автомата можно считать количество уже произведенных операций вставки (удаления), и конечными состояниями считать {0,1,2,3}

Заключение

В рамках данной работы был сделан обзор существующих алгоритмов и структур данных, используемых в нечетком поиске на этапах фильтрации и сравнения слов. Разработаны и реализованы алгоритмы, учитывающие специфику задачи:

- малый порог разницы (<=3)
- короткие слова (<=31)
- современное оборудование (поддержка SSE4.2)

Список литературы

1. Гасфилд Д. Строки, деревья и последовательности в алгоритмах: Информатика и вычислительная биология. СПб.: Невский Диалект; БХВ-Петербург, 2003. 654 с.
2. Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.* 33, 1 (March 2001), pp. 31-88.
3. Edgar Chavez, Jose L. Marroquin, and Gonzalo Navarro. 2001. Fixed Queries Array: A Fast and Economical Data Structure for Proximity Searching. *Multimedia Tools Appl.* 14, 2 (June 2001), 113-135.
4. Ciaccia, Paolo; Patella, Marco; Zezula, Pavel. "M-tree An Efficient Access Method for Similarity Search in Metric Spaces". *Proceedings of the 23rd VLDB Conference Athens, Greece, 1997*. IBM Almaden Research Center: Very Large Databases Endowment Inc. pp. 426–435. p426. Retrieved 2010-09-07.
5. W. A. Burkhard and R. M. Keller. Some approaches to best-match file searching. *Commun. ACM* 16, 4 (April 1973), pp. 230-236.
6. Peter N. Yianilos. 1993. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces. 311-321.

Приложение 1. Спецификация системы

Формальная постановка задачи следующая: реализовать систему, которая реализует следующие 4 функции (функции вызываются последовательно):

- *StartQuery*
- *EndQuery*
- *MatchDocument*
- *GetNextAvailRes*

Рассмотрим каждую из них подробнее:

StartQuery - добавить запрос к множеству активных запросов, где запрос представляет собой набор строк длиной от 4 до 31 символа, используемую метрику и максимально допустимое отклонение в данной метрике. Количество слов в запросе не превышает пяти.

EndQuery - удалить запрос из множества активных запросов.

MatchDocument - добавить документ на проверку, где документ представляет собой набор слов длиной от 4 до 31 символа, а общее количество символов в документе (учитывая разделительные пробелы) не превышает 1000000. Проверка же представляет собой построение списка запросов из числа активных на момент добавления документа, для которых каждое слово имеет близкое к нему (в соответствующей метрике и с соответствующим пороговым значением) среди слов данного документа.

GetNextAvailRes() - получить результат (если таковой имеется) для какого-либо документа, данного системе. Фактически, эта функция позволяет эффективно реализовать проверку документов параллельно, так как не требуется выдавать ответ для конкретного документа по окончанию выполнения функции *MatchDocument*, а можно накопить набор документов для обработки и при вызове этой функции подождать окончания обработки какого-либо из документов и выдать соответствующий результат.

В системе должна быть реализована поддержка следующих метрик:

- 0-1 метрика [?]: то есть точное сравнение двух строк;
- Метрика Хэмминга [?]: количество несовпадающих символов в соответствующих позициях двух строк (определена для строк одинаковой длины);

- Метрика Левенштейна [?]: минимальное количество операций вставки, удаления и замены символов, необходимое для преобразования одной строки в другую.

Система должна быть реализована как разделяемая библиотека на языке c++ и выполняться под управлением OS Linux.