Классификация и экспериментальное исследование современных алгоритмов нечеткого словарного поиска*

© Бойцов Леонид Моисеевич

Яndex
itman@yandex.ru

Аннотация

Цель данной работы заключается в классификации и экспериментальном сравнении существующих алгоритмов нечеткого словарного поиска.

Нами были реализованы и протестированы алгоритм последовательного перебора, модификации n-граммный алгоритмов, trie-деревья, метрические деревья, kd-деревья, а также менее распространенные сигнатур–ные алгоритмы.

В отличие от большинства других работ мы тестируем не только скорость поиска в индексе, загруженном в память, но и скорость чисто дискового поиска, когда индекс считывается непосредственно с диска.

1 Введение

В настоящее время алгоритмы нечеткого поиска строк получили широкое распространение в системах автоматизации перевода [22], орфографических корректорах, программах распознавания печатного текста и даже в поисковых системах.

В общем случае нечеткий текстовый поиск подразумевает отыскание произвольных участков текста, но часто задачу можно свести к словарному поиску.

Так, например, индексация во многих современных поисковых системах и электронных каталогах документов основана на технологии инвертирования [20].

Процессы инвертирования и составления алфавитного указателя имеют много общего и основаны на выделении значимых ключевых слов и составлении списков вхождений ключевых слов в текст.

Полученную в результате такого преобразования структуру данных принято называть инвертированным индексом или инвертированным файлом, а список ключевых слов – словарем.

Труды 6ой Всероссийской научной конференции "Электронные библиотеки: перспективные методы и технологии, электронные коллекции" - RCDL2004, Пущино, Россия, 2004.

Поиск в инвертированном файле осуществляется в два этапа: сначала происходит выборка слов запроса из словаря, а затем считываются и обрабатываются соответствующие списки вхождений.

Большинство современных электронных библиотек – это коллекция документов, снабженных инвертированных индексом для быстрого доступа. Центральным звеном поисковых модулей является словарный поиск. Ошибки и искажения могут быть, как во вновь добавляемых в систему документах, так и в запросах пользователей, поэтому задача эффективного нечеткого словарного поиска возникает, как на этапе создания документа, так и на этапе поиска в уже проиндексированной коллекции.

В настоящее время большая часть работ в области нечеткого строкового поиска относится к поиску без предварительного индексирования, который в англоязычных работах часто называется on-line поиском [13,15,23,24,25,26,27]. Словарный поиск с предварительной индексацией (off-line поиск) является сравнительно малоизученным направлением.

Хотя разработано немало методов и алгоритмов таких, как n-граммная индексация, основанная на индексации фиксированной длины [17], различные модификации метрических деревьев [5], алгоритмы поиска в абстрактных метрических пространствах [6,7,28], trie-деревья (лучи) [3,21] практически невозможно найти работы, посвященные сравнительному анализу алгоритмов нечеткого словарного поиска.

2 Классификация и описание поисковых алгоритмов

2.1 Функции похожести строк

Функция похожести строк – это краеугольный камень нечеткого словарного поиска. Выбор подходящей функции похожести влияет не только на качество выборки и скорость поиска, но также и на сложность реализации индекса. Хорошая функция близости слов учитывает различные типы искажений, включая удаления, замены, вставки и транспозиции символов, а в идеале и похожесть звучания слов.

Одна из первых предложенных мер близости слов – это функция Левенштейна [4]. Расстояние Левенштейна равно минимальному числу элементарных операций редактирования, необходимых для преобразования одной строки в другую.

Для расстояния Левенштейна такой набор включает операции замены, вставки и удаления одного символа. В модификации расстояния редактирования, предложенной Дамерау [9], в множество элементарных операций включены транспозиции символов. При этом требуется, чтобы к транспонированным символам не применялись другие операции редактирования.

Определенное таким образом расстояние редактирования может быть вычислено с помощью метода динамического программирования [26]. Алгоритм имеет сложность $O(MN)$, где M и N – длины сравниваемых строк, а для нахождения значения расстояния требуется вычислить MN элементов так называемой матрицы динамического программирования.

Сложность вычисления расстояния Левенштейна–Дамерау имеет квадратичный порядок относительно размера строк. Значительное количество работ посвящено созданию более эффективных алгоритмов.

Предложенные процедуры вычисления можно условно разделить на две категории. Первую категорию составляют алгоритмы, см. напр. [24], использующие метод отсечений. В их основе лежит алгоритм динамического программирования, но для определения расстояния редактирования не требуется вычислять все MN элементов матрицы. Вторую категорию составляют алгоритмы, основанные на эффективном использовании битовых операций [15,27]. Более подробную информацию и детальный обзор алгоритмов on-line поиска можно найти в [18].

Другой подход к задаче ускорения вычисления расстояния редактирования заключается в выборе более легко вычислимой функции похожести.

Так, например, были предложены и хорошо исследованы различные модификации n-граммных расстояний [25], основанные на подсчете числа общих подстрок фиксированной длины. Нам также известны примеры функций близости, основанные на вычислении длины наибольшей общей подпоследовательности двух строк [14].

В настоящее время предложено множество альтернативных функций близости, но с нашей точки зрения расстояние Левенштейна–Дамерау наиболее точно соответствует интуитивному понятию похожести. Кроме того, можно обобщить функцию Левенштейна, чтобы она точнее оценивала фонетическую похожесть слов (см. раздел 4).

Далее в работе мы будем рассматривать алгоритмы, ищущие лишь относительного элементарного расстояния редактирования (без учета транспозиций и весов символьных преобразований).

Такой подход оправдан в случае, когда функция Левенштейна может быть использована в качестве фильтра. Если объем выборки, полученной с помощью простых алгоритмов, невелик, то для каждой найденной строки можно уточнить расстояние до поискового образца, используя более качественную и ресурсоемкую функцию.

2.2 Параметры классификации поисковых алгоритмов

Цель индексации списка слов – ускорение поиска по сходству. Под поиском по сходству подразумевается отыскание всех слов, для которых расстояние (в нашей работе расстояние Левенштейна) до поискового шаблона не превышает заданную величину.

Алгоритмы, которые позволяют отыскать все строки словаря в заданной окрестности поискового термина, мы будем называть детерминированными.

Поскольку понятие меры близости само определено неточно, не всегда имеет смысл выборка всех слов в заданной окрестности поискового шаблона. Существуют алгоритмы, которые находят большую часть, но не гарантируют нахождение всех строк. Такие алгоритмы мы будем называть рандомизированными. Типичным примером является поиск слов, имеющих то же значение функции soundex, что и искомое слово.

Довольно распространенным подходом к реализации алгоритмов рандомизированного поиска является индексация относительно значений нескольких хеш-функций. Каждая из хеш-функций преобразует слова в числовые значения.

Например, в алгоритме локально устойчивого хеширования [10] хеш-функции строятся так, что чем меньше расстояние между двумя словами, тем больше вероятность того, что значения хеш-функций на этих словах совпадают. Уменьшая или увеличивая количество хеш-функций, можно достичь желаемого соотношения скорости поиска и полноты выборки.

Рандомизированные алгоритмы весьма разнообразны. С одной стороны, они, как правило, используют индексацию на точное равенство, а потому весьма эффективны, а с другой стороны сильно отличаются по полноте и точности выборки. Это означает, что сравнивать такие алгоритмы следует в первую очередь относительно качества выборки, что не является целью нашей работы.

В дальнейшем мы будем рассматривать только детерминированные относительно функции Левенштейна методы поиска.

В случае тривиального индексирования, когда запрос обрабатывается методом последовательного перебора, не требуется никакого дополнительного преобразования. Недостаток такого подхода – низкая эффективность. Выделяя в строках общие элементы, можно использовать их для сокращения перебора.

Мы будем называть процесс выделения характерных элементов строк сэмплированием. Чаще всего используются следующие методы сэмплирования:

- сэмплирование подстрок: префиксов, суффик–сов или n-грамм;
- буквенное сэмплирование;
- метрическое сэмплирование.

Метрический метод сэмплирования заключается в вычислении вектора расстояний от заданной строки до образующих элементов. Образующие элементы могут быть выбраны из существующих строк словаря или сгенерированы.

В целях поиска сэмплы могут быть преобразованы. Так, например, на основании набора n-грамм могут быть построены, как инвертированные списки, так и частотные вектора. Следующие два вида преобразований сэмплов наиболее распространены:

- построение частотного вектора;
- построение сигнатурного вектора.

Для построения вектора требуется хеш-функция *H(s)*, отображающая сэмпл в целое число, которое трактуется как индекс элемента вектора.

Сигнатурный вектор – это битовый массив, *i*-ый элемент которого равен единице, если существует сэмпл s такой, что $f(s) = i$. Частотный вектор – это массив, *i*-ый элемент которого равен числу сэмплов *s* таких, что $f(s) = i$.

Осуществив преобразование строк в сэмплы, можно проиндексировать словарь для быстрого доступа. Для этого можно использовать следующие алгоритмы:

- trie-деревья [21];
- инвертированный индекс [20];
- координатные структуры, например, kd-деревья [6] и rd-деревья [11];
- “точный индекс”, используемый для поиска методом расширения выборки.

Комбинируя различные методы сэмплирования и индексирования, можно строить новые алгоритмы. Разнообразие существующих алгоритмов – это основная проблема, возникающая при их экспериментальной проверке.

Даже распространенные алгоритмы имеют множество модификаций, потому в следующих разделах приведены описания реализованных алгоритмов.

2.3 Сигнатурные алгоритмы

Нами был реализовано два сигнатурных алгоритма: хеширование по сигнатуре [1] и частотные trie-деревья. В основе обоих алгоритмов лежит буквенное сэмплирование.

В случае хеширования по сигнатуре сэмпл преобразуется в сигнатурный вектор, который можно рассматривать, как запись числа в двоичном представлении. Таким образом, хеш-функция *H(a)* однозначно определяет преобразование *F(w)* строки в целое число. *F(w)* является хеш-функцией и может быть использовано для индексации словаря.

Если удалить или добавить одну букву, то изменится не более одного бита сигнатуры. Может не измениться ни один бит, если удаленная буква встречается в слове более одного раза, либо в слове есть буква, отображаемая функцией *H(a)* в тот же самый индекс сигнатуры, что и удаленная буква.

Если заменить один символ, то изменится не более двух битов сигнатуры. В случае изменения двух битов, один бит обнуляется, а другой становится равным единице.

Несложно показать (см. [1]), что для выборки всех слов, отличающихся от искомого на одну операцию редактирования, требуется прочесть не более

$$1 + m + m^2/4 \quad (1)$$

списков, где m размер сигнатуры. Общее число списков равно 2^m, и, если m достаточно велико, то количество списков, просматриваемых в процессе поиска значительно меньше общего числа списков хеш-таблицы.

Практически все сигнатурные алгоритмы, в частности хеширование по сигнатуре, очень чувствительно к выбору параметров и хеш-функции *H(a)*.

Размер сигнатуры не должен быть очень большим или очень маленьким – в обоих случаях поиск не может осуществляться эффективно. Опыт показал, что 13 является оптимальным в смысле соотношения скорости поиска и объема индекса размером сигнатуры для словарей, содержащих от нескольких сот тысяч до нескольких миллионов записей. Для данного размера сигнатуры число просматриваемых списков не больше 56, а общее число списков равно 8191.

В самом первом приближении, предполагая, что размеры списков примерно одинаковы, можно считать, что при поиске с максимально допустимой операцией редактирования равной 1 требуется перебрать примерно $56/8191 \approx 0.007$ всех строк.

Несмотря на то, что предположение равномерности не вполне корректно, оценка на основе формулы (1) (см. раздел 3) верно отражает реальную эффективность хеширования по сигнатуре. Несложно также видеть, что число перебираемых в процессе поиска записей линейно зависит от размера словаря.

Скорость поиска в сигнатурном хеше существенно зависит от выбора хеш-функции *H(a)*. Рассмотрим следующую классическую хеш-функцию:

$$H(a) = (ASCII\_CODE(a) - base) \bmod m, \quad (2)$$

которая вычисляет разность ASCII кода буквы и некоторого начального значения по модулю размера сигнатуры. Такая функция проста, легко вычислима, а время поиска в сигнатурном хеше на ее основе в несколько раз меньше, чем для алгоритма последовательного перебора.

Предложенная функция, как показали эксперименты, довольно неравномерно

распределяет буквы на группы. Если заменить (2) равномерной хеш-функцией *H(a)*, для которой частоты появления единичных битов в различных элементах сигнатуры примерно одинаковы, то скорость поиска возрастает в два раза.

Чтобы построить такую хеш-функцию, необходимо знать частоты появления букв. К счастью, для словарей, содержащих не более 2-3 миллионов слов, частоты легко подсчитать. На основе частотной информации хеш-функция может быть сконструирована с помощью простого "жадного" алгоритма.

Хеширование по сигнатуре хорошо подходит для так называемого "дискового" поиска, когда страницы индекса выбираются непосредственно с диска, потому что в процессе поиска считывается относительно небольшое число списков, занимающих несколько последовательных дисковых страниц.

Для индексов, загружаемых в память целиком, можно применить более эффективный подход, индексируя вместо битовой сигнатуры частотный вектор. Вероятность появления слова с заданным частотным вектором значительно меньше, чем вероятность появления слова с заданной сигнатурой при условии, что сигнатурный и частотный вектор имеют одинаковый размер, а также строятся с помощью одной и той же хеш-функции *H(a)*. Именно поэтому структуры данных на основе частотных векторов обладают потенциально большей способностью к сокращению перебора.

Списки частотных векторов короче списков сигнатур, но число различных частотных векторов больше чем число сигнатурных. В отличие от хеширования по сигнатуре для индексации частотных векторов нельзя использовать хеш-таблицу, ключом которой является частотный вектор, потому что перебор элементов такой таблицы занимает слишком много времени.

Чтобы решить проблему эффективной выборки похожих частотных векторов, следует их проиндексировать. Мы рассматривали несколько вариантов индексов и пришли к выводу, что для этой цели лучше всего походят trie-деревья.

В trie-дереве строки с одинаковым префиксом располагаются в одном поддереве. Распространяя этот принцип на вектора, мы будем располагать в одном поддереве вектора, у которых совпадают элементы с номерами 1, 2, … , k, где $k > 0$.

На самом верхнем уровне группируются вектора с совпадающим первым элементом. Эти вектора составляют отдельное поддерево. Соответственно, на втором уровне дерева вектора с одинаковым первым элементом группируются по значению второго элемента, и т.д. Для узлов, имеющих ровно одного потомка, применяется алгоритм сжатия путей – такие узлы объединяются в один.

Важным отличием от метода хеширования по сигнатуре является использование простой хеш-функции (2). Мы предполагаем, что частотно-равномерная хеш-функция обеспечивает более высокую скорость поиска, но не проверяли это экспериментально.

Пусть u и v – частотные вектора. Введем обозначения:

$$S_+(u,v) = \sum_{u_i - v_i > 0} u_i - v_i$$

$$S_-(u,v) = \sum_{u_i - v_i < 0} -u_i + v_i$$

Обе суммы S_+ и S_- неотрицательны. Поиск в частотном trie-дереве основывается на следующих двух свойствах:

- Значение S_+ *(*S_-*)*, вычисленное для первых k элементов векторов u и v, меньше либо равно S_+ *(*S_-*)*, вычисленного для векторов целиком;
- Минимально возможное расстояние редактирования между строками, с частот–ными векторами u и v равно $\min(S_+, S_-)$.

С помощью метода математической индукции можно также показать, что алгоритм поиска частотных векторов в trie-дереве (но не алгоритм в целом) имеет сложность:

$$O(m^{2k}),$$

где k – максимально допустимое расстояние редактирования, а m – длина частотного вектора.

Общее количество слов, считываемых в процессе поиска, зависит от вероятностей появления слов с заданным частотным вектором. Несложно показать, что среднее время поиска, как и в случае хеширования по сигнатуре, пропорционально числу проиндексированных записей.

Алгоритмы, базирующиеся на частотных и сигнатурных векторах, неоднократно рассматривались другими авторами, но, как правило, в контексте поиска подстрок (см. напр. [12]), а не слов целиком.

2.4 Алгоритмы n-граммной индексации

Мы реализовали два классических алгоритма n-граммной индексации, основанных на инвертировании списков n-грамм и два алгоритма индексации частотных n-граммных векторов: на основе kd- и trie-деревьев.

Рассмотрим сначала классические алгоритмы n-грамм, основанные на инвертировании. Уже более 30 лет (мы рекомендуем посетить веб-страницу [19] для детального изучения вопроса) n-граммная индексация используется в области информацион–ного поиска.

Словарная n-граммная индексация основана на следующем свойстве : если слово u получается из слова w в результате не более чем k элементарных операций редактирования (за исключением перестановок символов), то при любом представлении u в виде конкатенации из k+1-ой строки, одна из строк такого представления будет точной подстрокой w.

Это свойство можно усилить, заметив, что среди подстрок представления существует такая, что разность между её позицией в строках *w* и *u* не больше *k*.

Таким образом, задача поиска сводится к задаче выборки всех слов, содержащих заданную подстроку. Для решения этой задачи удобно использовать инвертирование относительно набора n-грамм слова.

Инвертированный файл позволяет эффективно определять список слов, содержащих заданную n-грамму. Если требуется сделать выборку по подстроке, которая длиннее *n* символов, то нужно считать один или более списков n-грамм этой подстроки и выполнить операцию пересечения списков. Если искомая подстрока короче *n* символов, то требуется считать списки всех n-грамм, у которых начало совпадает с искомой подстрокой и выполнить операцию объединения.

Для всех элементов результирующих списков вычисляется расстояние до поискового образца, и оставляются лишь те строки, у которых расстояние не превышает допустимое.

Согласно нашему замечанию искомая подстрока не может начинаться в произвольной позиции, поэтому можно ускорить поиск, если инвертировать относительно пары (n-грамма, позиция n-граммы в слове) и считывать списки только тех n-грамм, у которых позиция отличается от позиции в исходном слове не более чем на *k*. Описанный алгоритм мы будем называть первой модификацией n-грамм.

Если вместо одного файла, создавать несколько n-граммных индексов, в каждом из которых хранятся слова одной длины, то можно достичь большей эффективности при поиске в памяти. Такой алгоритм мы будем называть второй модификацией n-грамм.

Предположим, что средняя вероятность появления n-граммы в слове равна *p*. Тогда математическое ожидание длины списка вхождений n-грамм равно *Np*, где *N* – число записей в словаре.

С одной стороны, при поиске строки *w* длины *m* требуется считать списки не более *m* n-грамм. С другой стороны, мы никогда не считываем меньше *k* списков n-грамм. Время считывания одного списка пропорционально его размеру. Таким образом, в среднем сложность алгоритма равна:

$$O(pN),$$

а скорость поиска линейно зависит от числа проиндексированных слов.

Мы также реализовали два алгоритма, основанные на индексации частотных векторов n-грамм: kd-деревья и частотные n-граммные trie-деревья. Частотные вектора n-грамм строятся аналогично частным буквенным векторам с помощью хеш-функции, отображающей n-граммы в целые числа.

Kd-дерево [6] – это бинарное дерево, в котором каждый узел задает разбиение пространства на два подпространства. Одноразмерное kd-дерево – это обычное бинарное дерево: в левом поддереве узла находятся элементы с меньшими, чем в корне значениями, в правом – с большими.

Пусть размерность kd-дерева равна $m > 1$. При добавлении нового вектора в поддерево уровня *i* выбирается дискриминирующая размерность *d*. Как правило, $d = i \bmod m$.

Если *i*-ый элемент добавляемого вектора меньше либо равен значения *i*-го элемента вектора узла, то элемент добавляется в левое поддерево. В противном случае новый вектор добавляется в правое поддерево.

Как мы уже отмечали выше, каждый узел разбивает пространство на два подпространства. Если расстояние от поискового вектора до левого (правого) подпространства превышает максимально допустимое, то можно не продолжать поиск в левом (правом) поддереве.

После заполнения kd-дерева осуществляется его балансировка. Поскольку размеры словаря не очень велики: до 3 млн. записей, операция не является ресурсоемкой.

Известно [6,7], что среднее время поиска в сбалансированном kd-дереве равно $O(\log N)$, где *N* – число векторов. Время поиска в худшем случае имеет порядок $O(N^{1-1/m} + M)$, где *M* – число векторов в выборке.

Алгоритм поиска векторов в kd-дереве сублинеен, но алгоритм словарного поиска в целом на основе kd-деревьев имеет линейный рост времени поиска в зависимости от размера словаря, потому что количество слов, имеющих заданный частотный вектор, примерно пропорционально числу записей словаря.

Реализация частотного trie-дерева для индексации векторов n-грамм полностью аналогична реализации trie-дерева для индексации буквенных векторов. Отличается лишь условие ограничения перебора поиска в дереве.

Введем обозначение:

$$Dist(u,v,l) = \sum_{i \le l} | u_i - v_i |,$$

где *u* и *v* – частотные n-граммные вектора. Тогда узел уровня *i* исключается из дальнейшего рассмотрения, если $Dist(u,v,i)$ больше чем $k*n$, где *k* – максимально допустимое расстояние редакти–рования, а *n* – размер n-граммы.

2.5 Строковые trie-деревья

Мы использовали алгоритмы, описанные в [21]. На самом верхнем уровне группируются слова с одинаковой первой буквой. Эти слова составляют отдельное поддерево. Соответственно, на втором уровне дерева слова группируются по значению второй буквы, и т.д. Для узлов, имеющих ровно одного потомка, применяется алгоритм сжатия путей – такие узлы объединяются в один.

В процессе спуска по дереву происходит вычисление матрицы динамического программирования. Для каждой добавляемой при

спуске букве необходимо вычислить новый столбец матрицы (см. раздел 4).

Если в процессе поиска в последнем столбце (или в последних двух столбцах, если мы считаем транспозицию одной операцией редактирования) нет элементов меньших k, где k – максимально допустимое расстояние редактирования, то поддеревья текущего узла можно исключить из дальнейшего поиска.

Согласно [21] алгоритм поиска имеет сложность:

$$O(\Sigma^k),$$

где Σ – размер алфавита.

2.6 Алгоритм расширения выборки

Предположим, что слово u отличается от слова v ровно на одну операцию редактирования. Если построить множество всех слов, получающихся из u в результате одной вставки, замены или удаления символа, то полученное множество будет содержать v.

Это свойство можно использовать для сведения нечеткого поиска к точной выборке. Преимущество алгоритма заключается в том, что время поиска практически не увеличивается с ростом числа записей в словаре. Эксперименты показали, что для индекса загруженного в память и максимально допустимого расстояния редактирования равного единице – это самый быстрый алгоритм.

Основной недостаток алгоритма заключается в том, что он практически не применим для поиска с максимально допустимым расстоянием k большим единицы.

Несложно видеть, что сложность алгоритма равна: $O((\Sigma n)^k)$, где Σ – размер алфавита, а n – размер слова. И если для поиска с $k = 1$ требуется выполнить 300-500 “точных” запросов к словарю, то для $k = 2$ требуется не менее 10000 запросов.

Следует отметить, что в некоторых случаях можно ограничить класс операций замен и достичь приемлемого сочетания полноты и качества поиска.

2.7 Алгоритм последовательного перебора

При последовательном переборе строки считываются последовательно и сравниваются непосредственно с поисковым образцом. Для сравнения строк используются битовые алгоритмы agrep [27]. Наш выбор обуславливается высокой эффективностью этих алгоритмов.

Несмотря на то, что данный алгоритм работает медленно далеко не все реализованные алгоритмы, как показали эксперименты, намного эффективнее последовательного перебора. В частности для максимально допустимого расстояния редактирования равного двум многие алгоритмы оказываются медленнее в случае чисто дискового поиска.

2.8 Метрические деревья

Алгоритмы индексирования объектов в общих метрических пространствах весьма многочисленны. Метрическими деревьями мы будем называть только алгоритмы, использующие для индексирования набор расстояний до некоторых образующих точек.

Различные алгоритмы метрических деревьев, в частности vp-деревья [28] используют разбиение пространства на подпространства в зависимости от расстояния элементов подпространства до образующих точек. Для практически целей пригодны алгоритмы, размер индекса которых линейно зависит от числа проиндексированных записей.

При условии линейности индекса и наличии целочисленной функции расстояния самый быстрый поиск обеспечивают деревья Бёрхард-Келлера (БК-деревья) [5].

БК-деревья строятся следующим образом. Случайным образом выбирается корень дерева. Все множество элементов разбивается на поддеревья. Элемент w размещается в поддереве номер k, если расстояние редактирования от w до корня равно k. Для каждого поддерева выбирается случайным образом новый корневой элемент, и процесс индексации повторяется рекурсивно.

Поиск в БК-дереве основывается на неравенстве треугольника. Несложно видеть, что если расстояние от поискового образца до корня дерева равно p, то искомый элемент может находиться только в поддеревьях с номерами от $p – k$ до $p + k$, где k – максимально допустимое расстояние редактирования.

БК-деревья сублинейны. Согласно [5] среднее время поиска имеет порядок:

$$O(n^\alpha), \alpha < 1$$

На основании результатов экспериментов с помощью метода наименьших квадратов авторами [5] было показано, что $\alpha \approx 0.65$.

3. Экспериментальные данные

3.1 Условия эксперимента

В качестве рабочей платформы используется компьютер Pentium с тактовой частотой 1ГГц (512 Мб оперативной памяти), работающий под управлением операционной системы Линукс (ядро 2.2 или 2.4). В качестве языка программирования используется C++, компилятор gcc. Программы компилируются со вторым уровнем оптимизации.

Каждый алгоритм имеет две реализации: одна для словаря, индекс которого загружен в оперативную память, другая для словаря, индекс которого считывается с диска. В последнем случае все дисковые кеши очищаются перед каждой поисковой операцией. Для очистки кеша используется комбинированный метод чтения файла и вызов специальных функций ядра.

Сначала индекс создается в оперативной памяти, а только потом “сбрасывается” на диск. Созданная таким образом дисковая структура данных

оптимизирована для чтения, но не для быстрой модификации.

Доступ к данным на диске происходит по значению смещения в файле. Исключением является выборка указателей по заданной строке, для которой требуется хеш-индекс. Строковой хеш-индекс реализован с помощью библиотеки Berkley DB 4.1.

Индексируемые словари – это синтетические коллекции, созданные на основе словаря английских слов размером около 100 тыс. записей.

Для анализа роста времени поиска, размера индекса, и т.д. в разрезе количества индексируемых терминов мы используем несколько коллекций размером 100, 200, 400, 800, 1600 и 3200 тыс. слов.

Программа генерации словаря вычисляет частотное распределение длин слов, а также частоты появления букв в заданной позиции при условии, что букве предшествует заданная подстрока длины 3. Собранная статистика используется для синтеза словаря.

Как показали тесты средние значения времени поиска в реальном и синтетическом словаре такого же размера отличаются менее чем на 10%.

Наборы поисковых слов также генерируются автоматически. Размер тестового набора варьируется от 100 до 1000 слов.

Важным элементом эксперимента является функциональное тестирование, которое предшествует тестам. Суть тестирования заключается в генерации слов, получающихся из заданного в результате одной либо двух операции редактирования и поиске полученной строки в словаре.

Таблица 1 Время поиска в памяти (мс) k = 1

	1	2	4	8	16	32
расш. выборки	1.0	1.0	1.0	1.0	1.0	1.0
n-грамм (2)	0.2	0.4	0.7	1.4	2.6	4.9
част. trie-дерево	0.6	0.8	1.2	1.7	2.9	5.3
kd-дерево	0.9	1.4	2.3	4.1	7.1	12.9
хеш. сигнат.	1.4	1.9	2.8	4.5	8.7	17.2
n-грамм (1)	0.5	0.9	1.9	3.9	8.4	18.0
trie-дерево	18.5	19.7	20.1	20.5	20.9	21.3
метр. дерево	4.2	6.3	10.8	19.4	27.1	41.5
n-грамм част. trie	21.7	37.5	67.9	126.8	242.9	482.9
посл. перебор	43.4	87.1	174.4	350.7	709.9	1430.0

3.2 Сравнительный анализ результатов эксперимента

Несмотря на то, что мы старались оптимизировать все программные модули, отдельные алгоритмы могли быть реализованы несколько эффективнее. Поэтому важную роль при сравнении алгоритмов играет не только абсолютное время поиска, но и его рост в зависимости от числа записей в словаре.

Для максимально допустимого расстояния редактирования $k = 1$ самым быстрым поиском в памяти является алгоритм расширение выборки (см. табл. 1 – в первой строке размер словаря указан в тысячах записей). Алгоритм сублинеен, а время поиска составляет примерно 1 мс для словарей всех размеров.

Из экспериментальных данных можно видеть, что частотное trie-дерево и kd-дерево имеют примерно тот же порядок роста. Несмотря на то, что время поиска этих алгоритмов на больших объемах пропорционально числу записей в словаре, на словарях средних объемов и $k = 1$ доминирует сублинейная составляющая времени поиска.

Таблица 2 Время поиска в памяти (мс) k = 2

	1	2	4	8	16	32
част. trie-дерево	4.1	6.6	11.0	19.3	35.7	70.4
n-грамм (2)	2.6	5.0	9.6	18.8	37.3	74.8
хеш. сигнат.	4.7	8.4	15.8	30.8	64.3	134.0
trie-дерево	124.8	144.4	165.6	191.5	194.5	209.7
n-грамм (1)	6.4	13.5	27.9	59.3	126.9	267.9
kd-дерево	8.9	17.2	30.9	56.5	164.8	308.7
метр. дерево	30.1	49.2	103.3	192.1	342.5	539.2
расш. выборки	1067.4	1064.3	1063.2	1068.2	1116.3	1083.5
посл. перебор	70.1	140.8	283.0	569.4	1145.7	2305.4
n-грамм част. trie	103.0	190.3	373.8	702.3	1387.6	2816.8

С увеличение максимально допустимого расстояния редактирования ситуация меняется (табл. 2): только trie-деревья, метрические деревья и алгоритма расширения выборки демонстрируют сублинейный рост времени поиска.

Еще заметнее разница между сублинейными и не сублинейными алгоритмами видна из табл. 3, в которой приведена статистика отношения числа вычислений расстояния редактирования по отношению к общему числу записей в индексе.

Таблица 3 Эффективность отбора индекса k = 2

	1	2	4	8	16	32
trie-дерево	0.137	0.078	0.041	0.021	0.010	0.005
n-грамм (2)	0.016	0.016	0.014	0.013	0.012	0.010
част. дерево	0.021	0.021	0.021	0.021	0.021	0.021
n-грамм (1)	0.045	0.045	0.046	0.047	0.048	0.049
kd-дерево	0.081	0.079	0.076	0.073	0.069	0.064
хеш. сигнат.	0.075	0.075	0.074	0.075	0.080	0.082
метр. дерево	0.209	0.165	0.170	0.148	0.129	0.093
расш. выборки	3.538	1.769	0.885	0.442	0.221	0.111
n-грамм част. trie	0.930	0.932	0.933	0.934	0.935	0.935
посл. перебор	1.000	1.000	1.000	1.000	1.000	1.000

Это отношение – показатель эффективности отбора индекса. Если количество вычислений операций редактирования, необходимых для завершения поисковой операции, линейно зависит от числа слов в индексе, то на больших словарях время поиска растет пропорционально числу записей словаря, даже если индекс локально сублинеен.

Таблица 4 Скорость дискового поиска (мс) k = 1

	1	2	4	8	16	32
n-грамм (1)	28	35	46	57	93	142
хеш. сигнат.	44	61	85	109	156	209
част. trie-дерево	44	41	70	78	126	212
метр. Дерево	40	51	128	223	364	540
n-грамм (2)	75	98	164	229	377	571
kd-дерево	43	67	117	221	443	727
расш. Выборки	282	403	501	592	851	866
trie-дерево	77	122	273	458	892	1266
n-грамм част. Trie	123	98	259	343	792	1613
посл. Перебор	105	147	310	551	1092	2142

Если для поиска в памяти большинство реализованных алгоритмов на 1-2 порядка быстрее последовательного перебора, то в случае дискового поиска выигрыш далеко не такой значительный (см. табл. 4 и 5)

Размер индекса – важная характеристика индекса. Особенно эта характеристика важна для карманных персональных компьютеров.

Таблица 5 Время дискового поиска (мс) k = 2

	1	2	4	8	16	32
n-грамм (1)	35	56	86	130	245	427
n-грамм (2)	86	132	183	244	355	581
част. trie-дерево	85	78	146	202	394	766
хеш. сигнат.	63	94	157	278	499	867
kd-дерево	53	88	162	317	764	1410
trie-дерево	188	216	304	478	1008	1741
метр. дерево	147	220	464	820	1465	2139
посл. перебор	116	206	429	778	1537	3013
n-грамм част. trie	209	278	521	864	1741	3325
расш. выборки	12638	15585	19259	23217	28060	31394

Алгоритм последовательного перебора (табл. 6) имеет в нашем случае минимальный размер индекса. Под индексом этого алгоритма мы понимаем размер плоского файла со списком слов.

Сигнатурные алгоритмы и kd-деревья также имеют небольшой размер индекса – на хранение индексной информации затрачивается менее 20% от общего объема индексного файла. Порядка 80% занимают сами строки словаря.

Менее компактны, trie-деревья и метрические деревья. И самые большие по размеру индексы у классических n-граммных алгоритмов, размер индекса которых примерно в четыре раза превышает размер данных.

4. Использование обобщенного расстояния редактирования и учет фонетической похожести

Как правило, похожие по звучанию слова также имеют похожие написания. Но так бывает далеко не всегда, и практически идентично звучащие слова могут отличаться более чем на одну букву, например, *децкий* (ошибочное написание) и *детский.*

Понятно, что в данном случае функция Левенштейна слишком неточно оценивает близость слов. Чтобы улучшить функцию расстояния необходимо:

1. расширить множество элементарных операций редактирования за счет замен произвольных непустых подстрок;
2. использовать веса для учета стоимости различных операций редактирования;

Таблица 6 Размер индексных структур (Кб)

	1	2	4	8	16	32
посл. перебор	931	1897	3880	7928	16211	33163
n-грамм част. Trie	1278	2367	4498	8698	17132	34219
kd-дерево	1422	2582	4800	9101	17642	34824
част. Trie-дерево	1635	2928	5329	9882	18711	36225
хеш. сигнат.	2238	3155	5258	9618	18550	36873
trie-дерево	1684	3329	6575	12944	25411	49592
метр. дерево	2103	4241	8568	17303	34961	70663
расш. выборки	3520	6592	12800	25664	52800	106496
n-грамм (1)	4587	9026	18158	36837	75125	153635
n-грамм (2)	8491	12930	22062	40741	79029	157539

Первая проблема, возникающая при обобщении расстояния Левенштейна, заключается в том, что обобщенная функция близости может не удовлетворять неравенству треугольника и, следовательно, не быть расстоянием. Это значит, что метрические деревья не могут быть использованы для индексации.

Вторая и наиболее серьезная проблема заключается в том, что алгоритм динамического программирования не походит для вычисления обобщенного расстояния. Более того, мы сомневаемся, что существуют алгоритмы с полиномиальным относительно размера слов временем вычисления такого расстояния.

Наиболее простым с нашей точки зрения путем решения данной проблемы является определение меры близости, как значения, вычисляемого с помощью алгоритма динамического программиро–вания, учитывающего замены произвольных пар строк. Это значение также часто называют стоимостью оптимального выравнивания строк.

Кроме того, следует выбирать веса элементарных замен так, чтобы функция близости удовлетворяла неравенству треугольника. Очевидно, необходимо, чтобы неравенство треугольника выполнялось на множестве элементарных замен. По-видимому, это условие является также и достаточным.

В качестве альтернативы функции обобщенного редактирования можно использовать преобразование слова в строку фонем с последующим вычислением расстояния Левенштейна [29]. Недостатком такого подхода является неустойчивость к опечаткам: удаление

одной буквы может привести к изменению двух и более фонем.

Вычисление обобщенного расстояния редактирования более ресурсоемкая задача, чем вычисление расстояния Левенштейна. Рассмотрим определяющего его рекуррентное соотношение:

$$D(i,j) = \min\begin{cases} D(i-1,j) + c_{вставки}, i > 0 \\ D(i,j-1) + c_{вставки}, j > 0 \\ D(a,b) + c_{замены}(u_a u_{a+1}..u_i, v_b v_{b+1}..v_j) \end{cases}$$

Первые две строки соотношения соответствуют вставке одного символа в строки *u* и *v*, соответственно. Последняя строка соответствует обобщенной операции замены. Поскольку длины подстрок могут быть больше единицы, возникает задача эффективного поиска нескольких подстрок одновременно.

Можно заметить, что не обязательно осуществлять такой поиск при вычислении каждого элемента матрицы динамического программирования. Гораздо эффективнее построить множество возможных замен **перед** ее вычислением.

Если организовать список замен, возможных в позиции *i* в виде trie-дерева, то обобщенное расстояние можно вычислить примерно за $O(NM \times K \log_2 |A|)$ операций, где *K* – максимальная длина строки множества элементарных замен, |*A*| – размер алфавита.

Как мы уже отмечали выше, для расстояния Левенштейна разработаны алгоритмы вычисления примерно линейной сложности. К сожалению, нам неизвестны аналогичные результаты для обобщенного расстояния, а также результаты, уточняющие вышеприведенную оценку сложности вычисления обобщенного расстояния редактирования. В частности, интересно насколько она может быть улучшена с помощью отсечений Юкконена [24].

Не менее сложная задача – индексирование с учетом обобщенного расстояния редактирования. Из реализованных нами алгоритмов, только trie-деревья, метрические деревья, а также метод расширения выборки могут без особой потери производительности быть использованы для ее решения. Все остальные рассмотренные алгоритмы можно использовать только лишь в качестве “фильтра”.

К сожалению, подход, основанный на фильтрации неэффективен, если набор элементарных операций редактирования содержит много пар с двух- и трехбуквенными строками.

5. Выводы

В данной работе представлена классификация алгоритмов нечеткого словарного поиска и результаты их экспериментального сравнения. Эти результаты позволяют сделать вывод о том, что современные методики словарного нечеткого поиска намного эффективнее алгоритма последовательного перебора. В случае индекса, загруженного в оперативную память, время поиска отличается на 1-2 порядка.

Наши эксперименты позволяют выделять явно непригодные для практического использования алгоритмы: индексация частотных n-граммных векторов в виде trie-дерева и алгоритм последовательного перебора.

Из наиболее эффективных алгоритмов следует отметить алгоритмы n-грамм, trie-деревьев, а также сигнатурные алгоритмы, которые обеспечивают хорошее соотношение между размером индекса и скоростью поиска.

Несмотря на наличие эффективных алгоритмов для off-line поиска с учетом расстояния Левенштейна, проблема off-line поиска с учетом более общей функции близости остается открытой.

Литература

[1] Л.М. Бойцов. Использование хеширования по сигнатуре для поиска по сходству. *Прикладная математика и информатика, ВМиК МГУ*, № 8 стр. 135-154, 2001.

[2] Р. Грэхем, Д. Кнут, О. Паташник. Конкретная математика, М. Мир 1998.

[3] Д. Кнут, Искусство программирования. 3-е изд. М. Издательский дом “Вильямс”, 2000

[4] В.И.Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. *Докл. АН СССР*, 163, 4, стр. 845-848, 1965.

[5] Proceedings of the 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98), R. Baeza-Yates and G. Navarro. Fast Approximate String Matching in a Dictionary, pages 14-22, 1998.

[6] J.L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. In *Communication of the ACM*, volume 18(9), 1975.

[7] Philippe Chanzy, Luc Devroye, Carlos Zamora-Cura. Analysis of range search for random k-d trees. In *Acta informatica*, volume 37, issue 4-5, pages 355 – 383, 2000.

[8] Proceedings of the 5th Latin American Symposium on Theoretical Informatics. E. Chávez, G. Navarro. A Metric Index for Approximate String Matching, pages 181-195, 2002.

[9] Fred J. Damerau. A technique for computer detection and correction of spelling errors. In *Communications of ACM*, volume 7(3), pages 171-176, 1964.

[10] Proceedings of the 25th International Conference on Very Large Databases. A. Gionis, P. Indyk, R. Motwani. Similarity Search in High Dimensions via Hashing, pages 518-529. 1999.

[11] Database research group papers web-page. Technical Report of University of Wisconsin.

J.M. Hellerstein, A. Pfeffer. The RD-tree: an index structure for sets.
http://db.cs.berkeley.edu/papers/
[12] Proceedings of the 27th International Conferenc on Very Large Databases. T. Kahveci, Ambuj K. Singh, An Efficient Index Structure for String Databases, pages 351-360. 2001.
[13] U. Masek, M. S. Peterson. A faster algorithm for computing string-edit distances. In *Journal of Computer and System Sciences*, volume 20(1), pages 785-807, 1980.
[14] E.W. Myers. An O(ND) Difference Algorithm and Its Variations. In *Algorithmica*,volume 2(1), pages 251-266, 1986.
[15] E.W. Myers. A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Programming, In *Journal of the ACM (JACM)*, volume 46(3), pages 395 – 415, 1998.
[16] G. Navarro, R. Baeza-Yates, E. Sutinen, J. Tarhio. Indexing Methods for Approximate String Matching. In *IEEE Data Engineering Bulletin*, volume 24(4), pages 19-27, 2001.
[17] G. Navarro, R. Baeza-Yates. A Practical q-Gram Index for Text Retrieval Allowing Errors. In *CLEI Electronic Journal*, volume 1(2), 1998, http://www.clei.cl.
[18] G. Navarro. A Guided Tour to Approximate String Matching. In *ACM Computing Surveys*. Volume 33(1), pages 31-88, 2001
[19] Web site of the Computer science department of Maryland University.
Research on N-Grams in Information Retrieval.
http://www.cs.umbc.edu/ngram/
[20] C.J. van Rijsbergen. Information Retrieval, 1979.
The homepage of C.J. Rijsbergen.
http://www.dcs.gla.ac.uk/Keith/Preface.html
[21] H. Shang, T.H. Merret. Tries for Approximate String Matching. In *IEEE Transactions on Knowledge and Data Engineering*, volume 8(4), pages 540 – 547, 1996.
[22] Trados, computer aided translation software.
http://www.trados.com/.
[23] E. Ukkonen. Algorithms for approximate string matching. In *Information and Control*, volume (64), pages 100-118, 1985.
[24] E. Ukkonen. Finding approximate patterns in strings, O(k * n) time. In *Journal of Algorithms* volume 6, pages 132-137, 1985.
[25] E. Ukkonen. Approximate String Matching with q-Grams and maximal matches. In *Theoretical Computer Science*, volume 92(1), pages 191-211, 1992.
[26] R.A. Wagner and M.J. Fisher. The String to String Correction Problem. In *Journal of the ACM*, volume 21(1), pages 168-173, 1974.
[27] S. Wu, U. Manber. Fast Text Searching with Errors. In *Communications of the ACM*, volume 35 pages 83-91, 1992.
[28] Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms P.N. Yianilos. Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces, pages 311-321, 1993.
[29] Proceedings of the 19th annual international ACM SIGIR conference on Research and development in information retrieval. J. Zobel, P. Dart, Phonetic String Matching: Lessons from Information Retrieval, pages 166 – 172, 1996.

Classification and Experimental Comparision of Modern Dictionary Fuzzy Search Algorithms

Leonid Boitsov

The primary goal of this paper is to provide a taxonomy of modern dictionary (the so called off-line) fuzzy search algorithms as well as results of their experimental comparison.

Among implemented algorithms are agrep sequential search algorithm, four modifications of n-gram indexing algorithm, tries, kd-trees, metric tress and less common signature algorithms: signature hashes and frequency-vector tries.

Unlike most other papers we tested not only in-memory indexes but also indexes stored on disc. Moreover, in the latter case we cleared all disc caches before every search operation.

♣ Мы хотим выразить благодарность Максиму Губину, Гонцало Наварро и Джастину Зобелю, оказавших неоценимую помощь при написании данной работы.