электронный научно-технический журнал
ИНЖЕНЕРНЫЙ ВЕСТНИК
Издатель ФГБОУ ВПО "МГТУ им. Н.Э. Баумана". Эл No. ФС77-51036. ISSN 2307-0595

Особенности алгоритмов нечёткого поиска

 # 12, декабрь 2014

Желудков А. В., Макаров Д. В., Фадеев П. В.

УДК: 004.421.6

Россия, МГТУ им. Н.Э. Баумана
zhantonv@gmail.com

Постановка задачи и определение области применения.

Задача анализа алгоритмов нечёткого поиска на сегодняшний момент актуальна, так как область применения данных алгоритмов невероятно велика и разнообразна. Начнём с распознавания рукописных символов [1], которое с массовым распространением устройств с сенсорным экраном активно используется для обеспечения удобства ввода. Введённый символ преобразуется в комбинацию цифр в зависимости от последовательности произведённых жестов, и полученная комбинация сравнивается со значениями, заранее известными для всех символов используемого алфавита, записанными в таблицу. Символ, для которого совпадение будет самым полным, и считается распознанным. Именно для определения полноты совпадения и используются алгоритмы нечёткого поиска, поскольку распознанные значения могут отличаться от заложенных в таблице для некого конкретного символа. Следующая область, где данные алгоритмы успешно применяются, это формы заполнения информации на сайтах и полноценные поисковые системы вроде Google или Yandex. Например, такие алгоритмы используются для функций наподобие «Возможно вы имели в виду …» в тех же поисковых системах и для обнаружения опечаток в полях ввода программ. Эта возможность оказывается особенно полезной при составлении договоров для заключения сделок [2], где одна лишняя буква или символ могут впоследствии стать решающими при разрешении споров или иных конфликтных ситуаций между сторонами. Также данные алгоритмы активно используются в биоинформатике для сравнения генов, белков, хромосом [3], для работы с базами данных в системах мониторинга лесопожарной обстановки [4], обработки массивов данных в интересах кредитных организаций [5] и многих других областях. Именно поэтому так важно знать особенности основных применяемых на сегодняшний момент времени алгоритмов, чтобы для конкретной ситуации была возможность выбрать максимально эффективный из них, поскольку в большинстве задач это может сыграть значимую роль, а в случаях, схожих с мониторингом лесопожарной обстановки, даже стоить кому-то жизни. В данной статье анализируют-

ся наиболее используемые алгоритмы нечёткого поиска, и делается заключение о преимуществах и недостатках каждого из них. Основной целью является выявить наиболее оптимальные алгоритмы нечёткого поиска (с учётом их реализаций и модификаций) с точки зрения потребляемой памяти и времени, затрачиваемого на выполнение для среднего слова (8-12 символов), для дальнейшего использования в системах, требующих быстрого реагирования на поступающие изменения. Подобные системы базируются на работе с базами данных, в которых и производится неточный поиск какой-либо информации для определения значительности изменения того или иного фактора, т.е. находятся ли отклонения поступающих новых значений в пределах допустимой нормы или нет. Примером является система мониторинга лесопожарной обстановки, о которой упоминалось выше.

Анализ существующих алгоритмов.

Рассмотрим существующие алгоритмы нечёткого поиска и отберём самые актуальные на данный момент для дальнейшего анализа. Начнём с разработанного Робертом Расселом (Robert C. Russel) и Маргарет Кинг Оделл (Margaret King Odell) алгоритма Soundex [6]. Это один из алгоритмов сравнения двух строк по их звучанию. Он устанавливает одинаковый индекс для строк, имеющих схожее звучание в языке согласно заданной таблице схожих по звучанию символов и их сочетаний. Алгоритм обрёл большую популярность после того, как был опубликован в книге Дональда Кнута «Искусство программирования», том 1, Основные алгоритмы. Однако он имеет существенный недостаток, который не позволил ему распространиться во всём мировом сообществе, а именно данный алгоритм привязан к языку, на котором написаны анализируемые строки. Его использование в программах, предполагающих поддержку нескольких языков, требует отдельной таблицы схожих по звучанию групп символов для каждого языка, а для большинства языков определение таковой является нетривиальной задачей. Поэтому данный алгоритм используется сейчас в основном в англоговорящей среде, для которой подобная таблица уже существует.

Следующий из рассматриваемых алгоритмов — Алгоритм расширения выборки [7] — часто применяется в системах проверки орфографии. Он основан на сведении задачи о нечетком поиске к задаче о точном поиске. Данный метод подразумевает построение наиболее вероятных «неправильных» вариантов поискового шаблона. Т.е. строится множество всевозможных «ошибочных» слов, например, получающихся из исходного в результате одной операции редактирования, после чего построенные термины сравниваются на точное соответствие. Основной плюс данного алгоритма заключается в легкости его модификации для генерации «ошибочных» вариантов по произвольным правилам. Однако есть и минусы, главный из которых — большое число проверок для слов существенной длины, поскольку из них можно получить много «ошибочных» слов.

Ещё один из возможных алгоритмов для нечёткого поиска — это алгоритм, использующий код Хэмминга. Он давно и успешно применяется при кодировании и декодировании, позволяя восстановить утерянную при передаче информацию. Следует отметить, что,

несмотря на большую эффективность кодов Хемминга, они не лишены определенных недостатков. Линейные коды, как правило, хорошо справляются с редкими и большими опечатками. Однако, их эффективность при сравнении слов с частыми, но небольшими ошибками, менее высока. Также стоит обратить внимание на то, что в данном алгоритме присутствуют дополнительные затраты на кодирование информации.

Следующий из рассматриваемых алгоритмов не совсем подходит под поставленную задачу, но для полноты картины не упомянуть о нём всё же нельзя. Это алгоритм, использующий триангуляционные деревья, которые позволяют индексировать множества произвольной структуры при условии, что на них задана метрика. Существует довольно много различных модификаций данного метода, но все они не слишком эффективны в случае текстового поиска и чаще используются в базе данных изображений или других сложных объектов.

Далее рассмотрим алгоритм Вагнера-Фишера [8], который позволяет для двух строк найти расстояние Левенштейна — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую. Данный алгоритм имеет ряд значительных преимуществ перед всеми описанными до этого, а именно: относительно невысокую сложность реализации, возможность качественного сравнения схожести более чем двух строк, несколько вариантов реализации, которые можно использовать в зависимости от конфигурации системы, универсальность для всевозможных алфавитов. К недостаткам же можно отнести, что при перестановке местами слов или их частей получаются сравнительно большие расстояния. Значения между совершенно разными короткими словами оказываются маленькими, в то время как между похожими и длинными строками — значительными. Так же у данного алгоритма существует одна интересная модификация, которая позволяет находить расстояние Дамерау-Левенштейна. В нём к операциям вставки, удаления и замены символов, определенных в расстоянии Левенштейна, добавлена операция транспозиции (перестановки) символов. Фредерик Дамерау показал, что 80 % ошибок при наборе текста человеком являются транспозициями. Из-за всех перечисленных выше плюсов именно данная метрика, как и метрика поиска расстояния Левенштейна, наиболее часто применяется на практике, что можно увидеть из источников, используемых для определения области применения анализируемых алгоритмов нечёткого поиска. Именно по этим причинам в оставшейся части статьи будут наиболее подробно рассмотрены и исследованы эти два алгоритма и их модификации.

Алгоритм Вагнера-Фишера.

Рассмотрим формулу, по которой можно найти расстояние Левенштейна. Здесь и далее считается, что элементы строк нумеруются с первого, как принято в математике, а не с нулевого, как принято в языках C, C++, C#, Java, символа.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда редакционное расстояние (расстояние Левенштейна) $d(S_1, S_2)$ можно посчитать по следующей рекуррентной формуле (1)

$$d(S_1, S_2) = D(M, N) \tag{1}$$

где:

$$D(i,j) = \begin{cases} 0; i = 0,\ j = 0 \\ i;\ j = 0, i > 0 \\ j; i = 0,\ j > 0 \\ \min\begin{pmatrix} \mathrm{D(i, j-1)} + 1, \\ \mathrm{D(i-1, j)} + 1, \\ \mathrm{D(i-1, j-1)} + \mathrm{m(S_1[i], S_2[j])} \end{pmatrix}; \mathrm{j > 0, i > 0} \end{cases} \tag{2}$$

Где $m(a,b)$ равна нулю, если $a = b$ и единице в противном случае; $\min(a,b,c)$ возвращает наименьший из аргументов.

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

Справедливы следующие утверждения (3):

$$\begin{aligned} & d(S_1, S_2) \geq \big||S_1| - |S_2|\big| \\ & d(S_1, S_2) \leq \max(|S_1|, |S_2|) \\ & d(S_1, S_2) = 0 \Leftrightarrow S_1 = S_2 \end{aligned} \tag{3}$$

Рассмотрим формулу (2) более подробно. Очевидно, что редакционное расстояние между двумя пустыми строками равно нулю. Также очевидно то, что для получения пустой строки из строки длиной i, нужно совершить i операций удаления, а для получения строки длиной j из пустой, нужно произвести j операций вставки.

Осталось рассмотреть нетривиальный случай, когда обе строки непустые.

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

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

- Вставка символа и замена другого символа меняются местами.
- Замена символа с его последующим стиранием — неоптимально (лишняя замена).
- Стирание символа и замена другого символа меняются местами.

Пусть S_1 кончается на символ «a», S_2 кончается на символ «b». Есть три варианта:

Символ «a», на который кончается S_1, в какой-то момент был стёрт. Сделаем это стирание первой операцией. Итак, стёрли символ «a», после чего превратили первые i-1 символов S_1 в S_2 (на что потребовалось $D(i-1, j)$ операций), значит, всего потребовалось $D(i-1, j)+1$ операция.

Символ «b», на который кончается S_2, в какой-то момент был добавлен. Сделаем это добавление последней операцией. Превратили S_1 в первые j-1 символов S_2, после чего добавили «b». Аналогично предыдущему случаю, потребовалось $D(i-1, j)+1$ операций.

Оба предыдущих утверждения неверны. Если символы добавлялись справа от финального «a», то, чтобы сделать последним символом «b», нужно было или в какой-то момент добавить его (но тогда утверждение 2 было бы верно), либо заменить на него один из этих добавленных символов (что тоже невозможно, потому что добавление символа с его последующей заменой не оптимально). Значит, справа от финального «a» символы не добавлялись. Стирание самого финального «a» не проводилось, поскольку утверждение 1 неверно. Получается, что единственный способ изменения последнего символа — его замена. Заменять символ 2 или больше раз не оптимально. Значит,

1. Если $a=b$, то последний символ не изменялся. Поскольку он также не стирался и справа от него не приписывалось, он не влиял на наши действия, и, значит, было выполнено $D(i-1, j-1)$ операций.
2. Если $a\neq b$, то последний символ изменялся один раз. Сделаем эту замену первой. В дальнейшем, аналогично предыдущему случаю, нужно выполнить $D(i-1, j-1)$ операций, значит, всего потребуется $D(i-1, j-1)+1$ операций.

Было реализовано и исследовано 3 варианта этого алгоритма:

- Матричная реализация: строим полноценную матрицу D согласно формуле (2).
- Рекурсивная реализация: начинаем с последнего элемента матрицы D, в котором и содержится искомое расстояние, и рекурсивно находим все недостающие для расчётов элементы по формуле (2).
- Стековая реализация: будем сохранять следующие значение i, j и промежуточное результирующее значение в стеке. Тем самым можно отказаться от рекурсии и заменить её циклом, выполняемым, пока стек не окажется пустым.

Модифицированный алгоритм Вагнера-Фишера.

Также был реализован и исследован алгоритм поиска расстояния Дамерау-Левенштейна [9]. Для его вычисления необходимо внести в алгоритм Вагнера-Фишера

несколько изменений, а именно: независимо от выбранной реализации (ранее их предлагалось 3 варианта, далее будет рассмотрен матричный способ представления данных) работать теперь нужно не с двумя предыдущими строками матрицы, а с тремя. Появляется необходимость выполнять дополнительную проверку на использование транспозиции. Если она применялась, то при расчете расстояния необходимо учесть её стоимость. Данная модификация избавляет алгоритм поиска расстояния Левенштейна от основного недостатка, и расстояния между похожими длинными словами не оказываются теперь столь значительными.

Исследования реализаций и модификаций алгоритма Вагнера-Фишера.

Были реализованы и проведены исследования со всеми 3 вариантами реализации и 1 вариантом модификации (матричной реализации) данного алгоритма нечёткого поиска. На рисунках 1, 2, 3, 4 изображены графики зависимости времени работы алгоритмов от числа символов в строке. По горизонтальным осям отложено число символов в каждой из строк, по вертикальным — число тиков, потраченных на вычисление. Число тиков на графике является средним арифметическим от результатов 100 экспериментов на случайных строках.

Рисунок 1. Зависимость времени выполнения (в тиках) при матричной реализации алгоритма Вагнера-Фишера от длины входного слова

Рисунок 2. Зависимость времени выполнения (в тиках) при рекурсивной реализации алгоритма Вагнера-Фишера от длины входного слова

Рисунок 3. Зависимость времени выполнения (в тиках) при стековой реализации алгоритма Вагнера-Фишера от длины входного слова

Рисунок 4. Зависимость времени выполнения (в тиках) алгоритма поиска расстояния Дамерау-Левенштейна от длины входного слова

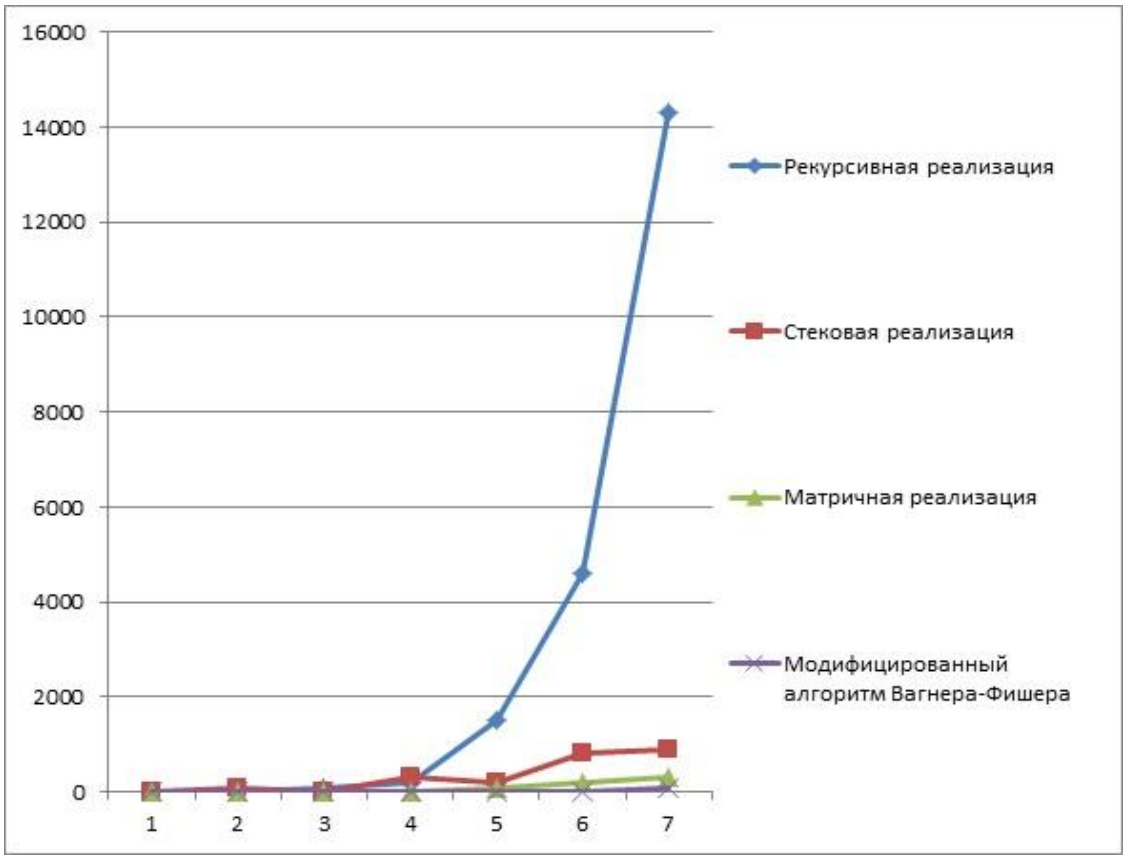

Рисунок 5. Сравнение по времени выполнения рекурсивной, матричной, стековой реализаций алгоритма Вагнера-Фишера и его модификации

В таблице 1 представлена зависимость требуемой памяти для каждого алгоритма от числа символов в строках:

Таблица 1. Сравнение по используемой памяти рекурсивной, матричной, стековой реализаций алгоритма Вагнера-Фишера и его модификации (для поиска расстояния Дамерау-Левенштейна)

Число символов в строках	Рекурсивная реализация [байт]	Стековая реализация [байт]	Матричная реализация [байт]	Модифицированный алгоритм Вагнера-Фишера [байт]
1	44	56	28	66
2	84	80	36	78
3	124	104	44	90
4	164	128	52	102
5	204	152	60	114
6	244	176	68	126
7	284	200	76	138
8	324	224	84	150
9	364	248	92	162
10	404	272	100	174
11	444	296	108	186
12	484	320	116	198
13	524	344	124	210
14	564	368	132	222
15	604	392	140	234

Обсуждение полученных результатов.

На первый взгляд получился достаточно неожиданный результат. Стековый вариант реализации, начиная с длины слов, равной 5, проигрывает по времени матричному, причем, чем больше длина слов, тем серьёзнее проигрыш. При длине слова 8 символов — разница уже в 10 раз, 10 символов — на 2 порядка, 12 символов — 3 порядка. И это только средний результат для 1-го слова. То есть при целом потоке запросов выигрыш будет измеряться в часах. Рекурсивный вариант проигрывает, поскольку происходит постоянный вызов процедуры, что приводит к сбросу конвейера ЦПУ и постоянному выделению памяти. Модификация алгоритма выигрывает за счёт ввода дополнительного действия — транспозиции, а, следовательно, и сокращению числа итераций по изменяемой строке. Но почему стековый вариант, который по своей сути является развёрнутой рекурсией, проигрывает по времени матричному? Ответ оказывается достаточно простым, но далеко не очевидным при первом рассмотрении задачи. Процесс работы стековой реализации алгоритма можно представить в виде троичного сбалансированного дерева. В отличие от рекурсивной реализации в нём отбрасываются ветви, число редакций в которых уже больше найденного минимума. Однако данное дерево включает в себя все возможные комбинации расположения строк и их модификаций относительно друг друга и, следовательно, получается больше переборов, чем в матричном варианте, где обрабатываются только текущая и предыдущая строки.

Анализ необходимой памяти для работы алгоритма показывает, что рекурсия, начиная уже с нетривиальных строк, хуже всех в данном показателе, так как на каждый следующий вызов выделяется память под передаваемые значения (если они являются ссылками или указателями, значит, память выделяется под ссылку или указатель) и локальные переменные. Стековая реализация стоит на втором месте, так как в стек кладётся чуть меньше параметров тех же типов, чем при рекурсивном вызове процедуры, а значит и суммарный его размер (по памяти) будет меньше. В матричной реализации, как уже ранее говорилось, обрабатывается только текущую и предыдущую строки, и поэтому это оказалась самая эффективная с точки зрения памяти реализация алгоритма Вагнера-Фишера и его модификации. Что же касается последнего из рассматриваемых случаев, то там память дополнительно затрачивается на работу с ещё одной строкой матрицы и некоторые оптимизационные, вспомогательные переменные.

Вывод и рекомендации.

Проанализировав полученные результаты, был сделан вывод, что из всех модификаций алгоритма Вагнера-Фишера, наиболее удачной является предложенная матричная реализация, которая решает задачу наиболее быстро и требует оптимального размера памяти, что немаловажно при использовании алгоритма для анализа данных в локальных датчиках или другом специализированном оборудовании. Если же затраты по памяти не являются приоритетным вопросом, то лучше использовать модификацию алгоритма (в матричной реализации) для поиска расстояния Дамерау-Левенштейна, поскольку она обрабатывает запросы быстрее всего и уменьшает требуемое количество действий на дальнейшее (если это необходимо) приведение сравниваемой строки к нужной. Хотелось бы подчеркнуть, что основное отличие проведённых исследований заключается в том, что наряду с рассмотрением существующих и использующихся на сегодняшний день алгоритмов акцентировалось внимание на вариантах реализации самого «привлекательного» с точки зрения практического применения и получаемого результата. При проведении подобных исследований этот момент обычно опускают, что мы считаем недопустимым. Данная позиция подтверждена полученными результатами, в которых преимущество оказалось у, казалось бы, достаточно «тяжеловесной» на первый взгляд реализации, а разница по времени между ними даже при сравнении одного слова средней длины (8-12 символов) составляет несколько порядков. Для систем, требующих быстрого реагирования на изменения анализируемых величин (например, система мониторинга лесопожарной обстановки), этот показатель является критичным, поэтому в зависимости от типа устройства (являющегося частью такой системы), требовательного или нетребовательного к памяти, следует использовать матричную реализацию или модификацию алгоритма Вагнера-Фишера (так же в матричной реализации) соответственно.

Список литературы

1. Didier Brun, Mouse Gesture Recognition URL: http://www.bytearray.org/?p=91 (дата обращения 25.11.2014)
2. Damerau–Levenshtein distance, URL: https://en.wikipedia.org/wiki/Damerau-Levenshtein_distance (дата обращения 25.11.2014)
3. Расстояние Левенштейна, URL: https://ru.wikipedia.org/wiki/Расстояние_Левенштейна (дата обращения 25.11.2014)
4. М.А. Сонькин, Ю.В. Лещик, «Применение алгоритмов нечеткого поиска в системах мониторинга лесопожарной обстановки», URL: http://www.lib.tpu.ru/fulltext/v/Bulletin_TPU/2012/v321/i5/20.pdf (дата обращения 25.11.2014)
5. Карахтанов Д.С, «Использование алгоритмов нечеткого поиска при решении задач обработки массивов данных в интересах кредитных организаций», URL: http://www.auditfin.com/fin/2010/2/11_02/11_02%20.pdf (дата обращения 25.11.2014)
6. Soundex метод нечёткого поиска, URL: https://ru.wikipedia.org/wiki/Soundex (дата обращения 25.11.2014)
7. Харитоненков А.В. «Поиск на неточное соответствие: коды Хемминга», http://www.jurnal.org/articles/2009/inf32.html (дата обращения 25.11.2014)
8. Задача о редакционном расстоянии, алгоритм Вагнера-Фишера, URL: http://neerc.ifmo.ru/wiki/index.php?title=Задача_о_редакционном_расстоянии,_алгоритм_Вагнера-Фишера (дата обращения 25.11.2014)
9. Расстояние Дамерау — Левенштейна, URL: https://ru.wikipedia.org/wiki/ Расстояние_Дамерау_—_Левенштейна (дата обращения 25.11.2014)