1 Обзор методов выборки данных по сходству.

1.1 Введение

За последние 30-40 лет количество информации, представленной в электронном виде, существенно возросло и продолжает возрастать экспоненциально. В связи с этим особую значимость получила задача полнотекстового поиска в этом постоянно увеличивающемся массиве данных. Первые информационно-поисковые системы (ИПС) появились более тридцати лет назад, и с тех произошли существенные изменения, как в поисковых алгоритмах, так и в техническом оснащении. Современные поисковые системы (ПС) "научились" автоматически собирать информацию в интернете, учитывать морфологические особенности и производить оценку значимости (релевантности или соответствия документа запросу) найденных документов. Тем не менее, такая проблема, как поиск по сходству, до сих пор остается по ряду причин недостаточно изученной.

Так, например, поиск по сходству занимает больше времени чем поиск на точное равенство. Кроме того, с одной стороны использование поиска по сходству увеличивает полноту выборки, то есть количество действительно релевантных документов, а с другой стороны увеличивает объем "шума" Однако в некоторых случаях он просто необходим, например, если пользователь не знает точного написания редкого слова или термина.

Поэтому целью данной работы является рассмотрение и сравнение методов поиска на неточное равенство, в том числе, методов поиска термина (ключевого слова) в словаре. Нужно отметить, что проблема поиска на неточное равенство очень многообразна: это может быть поиск в массиве текстовой информации, в базе данных ДНК, на множестве графов или изображений. Сразу хочу отметить, что в данной работе рассматриваются только текстовые поисковые системы. Текстовым документом буду называть список слов в некотором алфавите (это могут быть слова естественного языка или языка программирования), полагая при этом, что слова имеют относительно небольшую длину: 1-20 символов.

1.2 Поисковые системы и поиск по сходству

Как я уже отмечал ранее, объем документальных баз данных в последние годы значительно возрос. В настоящее время значительная часть электронных документов доступна в глобальной сети Интернет. С одной стороны Интернет обеспечивает почти мгновенный доступ к нужному документу (если известно его местоположение), а с другой стороны предоставляет почти неограниченные возможности для размещения данных. На сегодняшний день количество интернет-страниц и прочих электронных документов в сети Интернет настолько велико, что без систем поиска информации обнаружить необходимый ресурс практически невозможно. В настоящее время исследования в области ИПС нашли широкое применение в коммерческих поисковых системах.

Как показывает опыт, используя только поиск на точное соответствие, сложно найти нужную информацию. Как правило, различные поисковые термины (в основном слова естественных языков) могут использоваться в документах в различных грамматических формах. Поэтому современные ПС реализуют, как правило, поиск с учетом грамматических форм.

К сожалению, при вводе информации, например, при сканировании с бумажных носителей происходят ошибки, и данные оказываются, фактически, утерянными. Если пользователь осуществляет поиск в электронном словаре, то он может просто не знать точного написания слова. Таким образом, даже поиска с учетом грамматических форм бывает недостаточно. Неудивительно, что в последнее время наряду с различными методами классификации документов и распознавания смысловых аспектов запросов (если речь идет о естественных языках) в настоящее время интенсивно разрабатываются методы осуществления поиска по сходству. Актуальность задачи наглядно подтверждается количеством работ, написанных на эту тему.

Поиск может осуществляться в базах данных различного типа: в массивах документов, в мультимедийных системах или в хранилищах генетической информации. Текстовые информационно-поисковые системы занимают в этом списке лидирующие позиции и, несмотря на множество исследований и полученных результатов, это направление по прежнему активно развивается. С каждым днем все больше ученых занимается этой задачей и, видимо уже в самом ближайшем будущем, технология информационного поиска выйдет на принципиально новый уровень, характеризуемый как большими объемами хранимых данных, так и исключительно высокой производительностью и эффективностью поиска.

Исторически поисковые задачи в массиве документов разделяют на задачи информационного поиска (ИП, Information retrieval) и выборки данных (ВД, Data retrieval). Это разделение довольно условно, т.к. проблематика ИП тесно связана с задачами ВД. Исследования в области ИП имеют более концептуальный характер: ключевым моментом здесь является понятие **релевантности** документа поисковому запросу. К ВД относятся задачи построения и оптимизации индексов по скорости поиска и занимаемому дисковому пространству.

Поиск на неточное равенство, несомненно, находится на границе, потому что затрагивает два аспекта данной проблемы: поиск по неверно набранному ключевому слову (или подстроке ключевого слова) поискового шаблона, и нахождение документов релевантных запросу с заданной точностью.

В действительности, при поиске по ключевым словам, который часто называют **булевым поиском**, также используется понятие релевантности документа запросу, однако функция релевантности принимает только два значения: истина и ложь (0 и 1). В общем случае функция релевантности может принимать произвольные значения, например на отрезке от -1 до 1. При этом документ считается релевантным, если величина функции релевантности превышает некоторое наперед заданное число, называемое также пороговым значением.

Наиболее распространенные в настоящее время, см. например, [26], [4] гл. 12 и [15] поисковые методы используемые при создании ИПС можно разделить на две группы:

- поиск по ключевым словам (терминам)
- кластерные методы

В первую группу попадает, по-видимому, абсолютное большинство современных систем, в том числе и те ПС, которые обрабатывают запросы на естественных языках (Яндекс, Yahoo, и.т.д.). В процессе поиска такие системы, как правило, осуществляют выборку всех документов, содержащих хотя бы одно ключевое слово (грамматическую форму данного слова или синонима), а затем ранжируют эти документы по убыванию степени соответствия (релевантности) поисковому запросу.

Кластерные методы и векторные модели (ВМ). Основная идея ВМ, лежащая в основе поиска с использованием векторного представления документа и функцией релевантности (корреляции), состоит в следующем: каждый документ D отображается в вектор большой размерности $signD$, который я буду называть сигнатурой документа. Величина $k-$го элемента данного вектора зависит от частоты вхождения определенного компонента, например, подстроки длиной n символов, или по другому n-граммы (см. [29, 2.1-2.1.1]), в документ. При этом поисковое выражение Q также рассматривается как некий документ и для него находится сигнатура $sign(Q)$. Ключевым моментом построения модели является выбор функции корреляции $Corr(x, y)$ и порогового значения ϵ. В качестве функции корреляции может быть, например, использована функция, вычисляющая косинус угла сигнатур документов:

$$Corr(D_i, D_j) = \frac{(signD_i, signD_j)}{|signD_i||signD_j|}.$$

Поиск происходит следующим образом: последовательно считываются сигнатуры всех документов и вычисляются корреляционные значения $Corr(D, Q)$. В выборку попадают только те документы, для которых это значение превышает порог ϵ. Пользователю выдается список выбранных документов, ранжированных по убыванию величины $Corr(D_i, Q)$. Недостатком данного метода является необходимость в считывании сигнатур всех документов. Чтобы устранить этот недостаток используются алгоритмы кластеризации или объединения релевантных документов в группы (кластеры).

В основу разбиения документов на кластеры положено следующее наблюдение: близкие в смысле значения функции соответствия документы обычно релевантны одним и тем же поисковым запросам. Таким образом, разбив все документы на группы или кластеры и выбрав или построив в каждой группе характерного представителя: центроид кластера, можно сравнивать запрос не с каждым документом по отдельности, а сначала только с центроидами. Если центроид релевантен запросу, то нужно продолжать поиск внутри кластера, нет — перейти к рассмотрению другого множества документов.

К сожалению, в методах построении ВМ присутствует очень много эмпирических элементов таких, как выбор функции релевантности или порогового значения. Неправильно выбранное пороговое значение приводит к тому, что корреляционная функция нередко "забраковывает" подходящие слова (документы) и выбирает много лишнего. Нередко результат поиска — выборка объемом в несколько тысяч документов, многие из которых не содержат даже близких слов к словам в поисковом выражении.

Особенно это актуально для поиска на неточное равенство с использованием n-грамм (триад): в разделе 1.4 (стр. 24) приведен пример, когда опечатка в одном ключевом слове, приводит к тому, что значение корреляции между исходным словом и модифицированным меньше нуля.

Поиск по ключевым словам (терминам). Обратимся к более подробному описанию поиска по ключевым словам. Ввиду эффективности и простоты реализации данный метод получил наибольшее распространение. Основными методами организации индекса для поиска по ключевым словам являются:

- Инвертированные файлы (ИФ).
- Сигнатурные файлы (СФ).

ИФ является аналогом предметного указателя в конце книги. ИФ — это множество пар, ключевое слово, адрес вхождения ключевого слова в документ. Таким образом, ИФ содержит словарь ключевых слов и списки вхождений ключевых слов в документы. Каждому ключевому слову поставлен в соответствии свой список вхождений. Чтобы ускорить поиск, словарь может быть организован в виде дерева или хэш-индекса.

Сигнатурные файлы (СФ), как и ИФ, ориентированы на поиск по ключевым словам. Основное отличие заключается в представлении списка вхождений ключевых слов, поскольку СФ разрабатывались как альтернатива ИФ с более компактным индексом.

В простейшем варианте СФ выбирается хэш-функция $f(x)$, отображающая множество ключевых слов в отрезок целых чисел от 1 до n, и каждому документу ставится в соответствие битовый вектор (или по другому сигнатура документа, компоненты которого могут принимать значения только 0 и 1), у которого i-ый компонент равен 1, если в документе есть ключевое слово w такое, что $f(w) = i$, и нулю в противном случае. В процессе поиска по ключевому слову w последовательно перебираются все сигнатуры и находятся те, у которых компонент с номером $f(w)$ равен 1. Очевидно, что только такие документы могут содержать ключевое слово w, поэтому они последовательно сканируются на предмет проверки соответствия запросу.

Более подробное описание методов ИФ, СФ и ВМ приведено в разделе 1.4, где, в частности, приводятся данные из статьи Джастина Зоббеля ([17]) со сравнительной характеристикой методов ИФ и СФ.

Рассмотрим типы запросов ИПС. Типы поисковых выражений можно классифицировать по нескольким параметрам. Прежде всего их можно разделить на две большие группы: контекстно-зависимые запросы и контекстно-

независимые запросы. В случае контекстно-независимых запросов предполагается, что документ состоит из некоторых неделимых компонентов, в большинстве случаев: слов. Контекстно-зависимые запросы могут быть очень разнообразны, начиная с поиска некоторой подстроки в документе, и заканчивая условием на наличие некоторых ключевых слов в одном предложении, абзаце, и.т.д.

По способу задания поисковых запросов их можно разделить на **булевые или логические** и так называемые **ранжированные**. В случае булевого поиска пользователь задает логическое выражение на условие вхождения ключевых слов в документ. Например: документ содержит ключевые слова **двигатель** или **бензопровод** либо ключевое слово **автомобиль**. При поиске по ранжированным запросам текст запроса обычно записывается на естественном языке. ИПС разбирает запрос, выделяя в нем слова, или устойчивые выражения. Далее ИПС расширяет запрос синонимами терминов поискового запроса и производит поиск всех документов, содержащих хотя бы один термин. Найденные документы ранжируются по степени релевантности запросу.

Возможны различные варианты сопоставления ключевых слов запроса и документа:

- точное соответствие
- с учетом изменяемости слова, а также поиск по словоформам.
- по сходству

Возвращаясь еще раз к тому, что в данной работе понимается под поиском на неточное соответствие, или поиском по сходству, я хочу уточнить, что, везде где это не указано явно, под сходством я в первую очередь понимаю сходство терминов запроса и слов текстовых документов некоторой документальной базы данных.

Название первого пункта говорит само за себя: при поиске на точное равенство ищутся только те документы, в которые ключевое слово входит точно в той форме, которую указал в запросе пользователь.

К сожалению, поиск на точное соответствие, как я уже отмечал ранее, не позволяет найти слово, если в документе оно использовано в другом падеже, спряжении, и.т.д., и тем более тогда, когда в термине запроса или словах в документа имеются ошибки. Поэтому большинство современных “поисковиков” осуществляет поиск с учетом грамматических форм, но только некоторые умеют учитывать также возможные неточности в задании поисковых терминов. Наибольшее сложности возникают, разумеется, когда нужно вести поиск, как с учетом возможных ошибок в запросе, так и с учетом изменяемости слов.

Для того, чтобы можно было осуществлять поиск по сходству, словарь должен быть “устроен” соответствующим образом. Среди наиболее распространенных методов организации поиска по сходству в словаре можно выделить:

1. Сканирование словаря (с применением функции, вычисляющей близость слова поисковому образцу).

2. Расширение выборки (query extension).

3. Метод n-грамм или метод триад.

4. “Частичный” поиск с использованием хеширования.

5. Методы, использующие неравенство треугольника.

Рассмотрим основные идеи этих методов.

1. Данный метод применяется, например, в системе glimpse ([37]). Время поиска в словаре прямо пропорционально размеру словаря. В настоящее время разработано множество алгоритмов ([28], [30],[31],[36],[38]), которые достаточно эффективно реализуют данный метод. При этом процессорное время, затрачиваемое на обработку данных, мало по сравнению со скоростью считывания с диска, и если список ключевых слов загружен в оперативную память, оно составляет доли секунды для словаря, содержащего 200-300 тыс записей. В настоящее время намечается тенденция архивирования словаря и исходного массива документов ([19], [20],[21]). В связи с увеличивающимся разрывом между скоростью работы диска и мощностью процессора, затраты на поиск в упакованном индексе, при использовании специальных алгоритмов, невелики по сравнению со временем чтения с диска.

   Несмотря на довольно высокую эффективность некоторых поисковых программ, поиск на точное равенство осуществляется медленнее чем, скажем, для словаря, организованного в виде B-дерева. Кроме того, при занесении новой ссылки на документ, содержащий ключевое слово, словарь нужно целиком просканировать, чтобы определить местоположение ключевого слова, что замедляет построение индекса.

   Несомненным достоинством полного сканирования индекса является гибкость: можно без особых временных затрат, осуществлять поиск на точное и неточное равенство, а также по шаблону, являющемуся регулярным выражением.

2. Данный метод достаточно широко распространен, и его используют многие программы проверки орфографии, в числе которых бесплатно распространяемый ispell[1]. Основная идея заключается в том, чтобы свести задачу поиска на неточное равенство к задаче поиска на точное равенство путем построения наиболее вероятных “неправильных” вариантов поискового шаблона (обычно это шаблон, содержащий слова, для которых расстояние редактирования (Левенштайна) до исходного шаблона не больше 1) и поиска всех таких “ошибочных” вариантов в словаре. Кроме

[1]Бесплатно распространяемая программа проверки орфографии ispell может быть найдена, например, на следующем ftp-сервере:
ftp://sunsite.sut.ac.jp/pub/academic/russian-studies/Software/Ispell/

обычных ошибок, спел-чекер может сделать предположение об изменении формы (окончания, суффикса) слова.

Одним из вариантов расширения выборки является stemming (см. [27]), то есть отсечение окончаний и суффиксов ключевых слов в словаре и поисковом запросе. Для английского языка наиболее известен стеммер Портера ([25]).

Основным недостатком метода спел-чекера является его ориентированность на какой-либо конкретный язык. С увеличением размера алфавита скорость работы метода резко уменьшается. Явно проблематичен поиск по подстроке и по регулярным выражениям. Алгоритм расширения выборки, с моей точки зрения, удобно использовать лишь как вспомогательный механизм организации поиска.

3. Третий вариант — метод n-грамм (или триад), основан на следующем свойстве, используемом, например, программой agrep ([35]), или алгоритмом поиска в базе геномов FASTA ([8]): если слово u получается из слова w в результате не более k ошибок типа удаление, добавление и замена буквы, то если представить слово w в виде композиции $k+1$-го слова меньшей длины, то среди этих слов хотя бы одно будет подстрокой слова u.

   Для поиска можно применять, например, следующий алгоритм. Выделить в ключевых словах все подстроки небольшой длины (скажем 2 или 3) и организовать словарь в виде инвертированного файла, в котором словарем является множество подстрок, а документами — множество терминов словаря. Поскольку при небольших модификациях поискового термина, согласно приведенному выше свойству, исходный и модифицированный термины должны иметь общие подстроки длины 2 или 3, то такой индекс позволяет выделить списки слов потенциально содержащие нужный термин.

   При поиске на 1-2 ошибку, этот алгоритм работает быстрее методов полного сканирования и спел-чекера[2] и позволяет размещать данные на диске так, что они располагаются физически в смежных секторах, а скорость чтения оказывается близкой к скорости чтения большого плоского файла. Несмотря на этот очевидный плюс, индекс занимает в 10-20 раз больше места по сравнению с файлом, содержащим список ключевых слов! Кроме того, при поиске по двум ошибкам примерно в каждом из 5-10 случаев (для английского и русского языков) возникает потребность последовательно сканировать ключевые слова, потому что поисковый шаблон оказывается короче шести букв. Очевидно, что

[2]Если нужно организовать поиск только на одну ошибку, то алгоритм индексирования можно очевидным образом упростить, что приведет, в частности к уменьшению индекса. Для этого нужно помещать в словарь не все n-граммы, а только первую “половину” слова, укороченную на одну букву, и еще две n-граммы из “второй” половины. Очевидно, что в данном случае ошибка, заключающаяся в перестановке первой и последней (или второй и предпоследней) букв не будет обнаружена.

использование подстрок длиной в одну букву приводит к снижению скорости поиска.

В разделе 1.4.4 (на стр 40) о методе n-грамм и, в том числе об алгоритме Джона Вилбура[3], можно прочесть более детальный отчет.

4. Идея, лежащая в основе хеширующих методов, состоит в следующем: придумать отображение (хэш-функцию) слова, например, во множество чисел, или строк, "схватывающее" основные характеристики исходного слова, и устойчивое к большинству ошибок. Такая функция позволяет разбить все слова на группы и производить последовательный поиск внутри группы, вычисляемой по поисковому шаблону.

   Например, для разбиения на группы иногда используется хеширование по первым двум буквам слова. Другим очень известным примером является хэш-функция soundex, встроенная в такие промышленные СУБД, как Sybase, MS SQL Server и Oracle.

5. Будем при рассмотрении этого метода считать, что $\mathcal{U}$ — произвольное метризованное множество и не обязательно является множеством строк конечного алфавита. При этом исходный массив может быть не очень большим, но операция сравнения является очень дорогостоящей. В случае массива строк это не очень реалистичное предположение, однако в качестве объектов поиска может рассматриваться множество изображений или графов. Подход, описанный, например, в [11], позволяет построить дерево, существенно сокращающее количество операций сравнения при поиске. При этом количество рассматриваемых внутренних узлов дерева, довольно велико. Автор [11] рассматривает пример базы с 1млн слов фиксированной длины 60 и расстоянием Хэмминга[4] в качестве меры близости. При поиске по шаблону, заданному с точностью до 8 ошибок, а это примерно одна ошибка для слов длиной 8-10 символов, рассматривается около 10 процентов всех внутренних узлов дерева, и производится около 1000 (1%) операций сравнения. К сожалению, в случае массива коротких строк рассмотрение 10 процентов внутренних узлов влечет за собой большую дисковую активность и чтение большого количества несмежных страниц. (Подробнее данный алгоритм описан в разделе 1.4.4 стр. 42)

Возникает вполне закономерный вопрос, а можно ли существенно ускорить поиск на неточное равенство и уменьшить количество дисковых операций без существенного увеличения размера индекса? Можно ли организовать данные так, чтобы при поиске считывались страницы, располагающиеся на диске в соседних секторах?

Для поиска на точное равенство существуют алгоритмы ([16]) построения B-деревьев, которые компактны, позволяют считывать страницы с диска почти

[3]К сожалению, на момент написания данного обзора Джон Вилбур не завершил работу над статьей с описанием своего алгоритма.

[4]Расстояние Хэмминга между словами длиной k-бит равно количеству битов, в которых эти слова отличаются

последовательно, а количество считываемых страниц при выполнении поиска по префиксу имеет порядок $\mathcal{O}(\log n + N_{ocurr})$, где N_{ocurr} — количество слов, соответствующих поисковому запросу, а n — общее количество записей словаря.

Такие алгоритмы существуют, потому что все слова, имеющие одинаковые префиксы, располагаются подряд без пропусков в отсортированном списке ключевых слов. Если рассмотреть задачу поиска по подстроке, то почти очевидно, что как бы мы не располагали данные на диске, скорее всего, найдутся такие запросы, для выполнения которых нужно считывать страницы, не являющиеся смежными на диске, то есть с большим количеством позиционирований. Поэтому почти все существующие на сегодняшний день алгоритмы, позволяющие осуществлять поиск по подстроке, индексируют суффиксы (префиксы) строк. При этом налицо избыточность хранения данных.

По моему мнению, для поиска на неточное равенство подобный результат вряд ли достижим, и существенно ускорить поиск, в общем случае, без значительной избыточности представления данных, чтения несмежных страниц, или загрузки данных в оперативную память невозможно. Это, разумеется, не означает, что нужно отказываться от реализации подобной функциональности. В действительности, поиск по сходству, как показывает опыт использования программ коррекции орфографии, glimpse и agrep, можно успешно применять на персональной ЭВМ уже сегодня. Тем не менее, сложность реализации подобных проектов, в силу упомянутых выше причин, будет еще значительное время довольно высокой.

Я хочу еще раз отметить, что данная глава носит ознакомительный характер, в частности здесь не рассматриваются алгоритмы вычисления меры близости, которые долгое время были центральным моментом проблемы поиска по сходству, а также не рассматриваются детали организации основных поисковых структур: кластерных, сигнатурных и инвертированных файлов. Подробнее об этом можно прочесть в разделе 1.4.

1.3 Необходимые определения

В дальнейшем я буду постоянно пользоваться такими понятиями, как алфавит, шаблон, слово, мера близости (соответствия) слова шаблону и др., поэтому сперва я приведу определения, используемые в данной работе.

Определение 1.1 *Алфавитом* $\mathcal{A} = \{\Lambda, \alpha_1, \alpha_2, \ldots \alpha_\Sigma\}, \quad |\mathcal{A}| = \Sigma, \ \Sigma > 1$ *будем называть любое конечное множество символов, содержащее пробельный символ* Λ.

Определение 1.2 *Словом или строкой* w *в алфавите* $\mathcal{A}$ *длины* n*, будем называть последовательность символов алфавита* $\mathcal{A}$*. Будем также обозначать множество строк длины* n*, как* $\mathcal{A}^n$.

Замечание 1.1 *Введение пробельного символа позволяет рассматривать слова постоянной длины, так как к слову меньшей длины всегда можно добавить справа нужное количество пробелов.*

Определение 1.3 *Подсловом или подстрокой* $u = \alpha_i\alpha_{i+1}...\alpha_{i+k}$ *слова* $w = \alpha_1\alpha_2 \dots \alpha_i\alpha_{i+1} \dots \alpha_{i+k} \dots a_n$ *будем называть подпоследовательность символов слова* w *идущих подряд и без пропусков.*

Замечание 1.2 *Из определения следует, что произвольная подпоследовательность, вообще говоря, не считается подстрокой. Так* **пар** *является подстрокой слова* **паровоз**, *а* **про** — *не является.*

Определение 1.4 *Префиксом слова* $w = \alpha_1\alpha_2 \dots \alpha_n$ *будем называть подстроку вида* $\alpha_1\alpha_2 \dots \alpha_i$. *Другими словами, префикс является "началом" слова.*

Определение 1.5 *Аналогично суффиксом слова* $w = \alpha_1\alpha_2 \dots \alpha_n$ *будем называть подстроку вида* $\alpha_i \dots \alpha_{n-1}\alpha_n$.

Будем также считать, что рассматриваемое множество слов $\mathcal{U}$ может быть очень большим, но всегда конечным.

Определение 1.6 *Шаблоном* P *будем называть любое* **непустое**, **конечное** *множество слов, так называемых элементов шаблона.*

Замечание 1.3 *В силу того, что в работе рассматриваются только конечные множества слов ограниченной длины, не имеет смысла рассматривать бесконечные шаблоны, так как нас будет всегда интересовать только пересечение шаблона с основным множеством* $\mathcal{U}$.

Замечание 1.4 *В данной работе шаблон часто будет задаваться в виде некоторого условия на элементы шаблона, например, в виде регулярного выражения, или в виде условия, что строка должна содержать некоторую строку* w *в качестве подстроки. При этом сам шаблон будет задаваться в виде некоторой строки* u. *Будем в дальнейшем называть расширением шаблона построение его элементов.*

Рассмотрим примеры наиболее распространенных классов шаблонов:

- Самый простой пример шаблона — это произвольное слово. Несмотря на свою простоту, это наиболее распространенный поисковый шаблон.
- Еще один важный тип шаблона — регулярные выражения. Условимся, что символ %[5] означает произвольную, возможно пустую, подстроку, а _ — произвольный символ.

Например, строки **%кол%**, **__пол** являются регулярными выражениями. Легко видеть, что регулярные выражения являются шаблонами. Действительно, выражение **%кол%** задает множество всех слов, содержащих подстроку **кол**, а **__пол** — подмножество всех пятибуквенных слов, кончающихся на **пол**.

Кроме того, будем считать, что на множестве слов задана функция расстояния, которая не обязательно является метрикой.

Наиболее известной функцией расстояния, является функция Левенштайна, которая формализует интуитивное понятие об "ошибке".

[5]Выбор символов % и _ обусловлен использованием их в предикате **like** в языке SQL

Определение 1.7 *Функция Левенштайна равна минимальному количеству вставок (удалений, замен) букв, необходимых для преобразования слова w в v. В разделе 1.4 описывается более общая функция расстояния, которая вычисляет минимальное расстояние редактирования на основе положительной стоимостной функции δ, значения которой могут различаться для операций редактирования разных типов. В данном случае $\delta \equiv 1$, поэтому я буду обозначать функцию Левенштайна, как $edist_1(w, v)$.*

Замечание 1.5 *Несложно проверить, что $edist_1$ является метрикой. Функция $edist_1^*(w, v)$, определяемая, как минимальное количество перестановок двух букв, вставок, удалений, добавлений, которые преобразуют слово w в v, также является метрикой.*

Теперь можно дать окончательное определение меры близости слова и поискового шаблона:

Определение 1.8 *Расстоянием между словом u и шаблоном $P = \{w_1, w_2, \ldots, w_s\}$ будем называть минимальное расстояние в смысле метрики ϱ между u и элементами шаблона P:*

$$\varrho(u, P) = \min_i \varrho(u, w_i) \qquad (*)$$

Замечание 1.6 *В силу того, что шаблон конечен, минимум в (*) достигается на одном из элементов.*

Случай полного соответствия является частным, но не менее важным, случаем неточного равенства, когда $\varrho(u, P) = 0$.

1.4 Обзор литературы

Задача поиска на неточное равенство является слишком общей и многогранной, для ее решения не существует одной универсальной методики, поэтому ее необходимо рассматривать в контексте конкретных приложений таких как, текстовый и мультимедийный поиск, поиск гомологичных цепочек макромолекул в молекулярной биологии, поиск похожих графов, и.т.д. Несмотря на то, что в большинстве случаев для сравнения объектов используется функция похожести, нередко являющаяся метрикой, скорость ее вычисления, впрочем, как и объем дисковой памяти, занимаемой одним элементом (строкой, изображением, графом) могут очень сильно различаться. В связи с этим для различных типов приложений используются различные подходы к задаче оптимизации времени поиска. В случае текстового информационного поиска существуют алгоритмы, находящие все соответствия в тексте, с точностью до k ошибок в смысле функции Левенштайна, за $\mathcal{O}((k+1) \cdot (m+n))$ операций, где n — размер документа, или строки текста, а m — длина строки в поисковом запросе. Числа n, k, m, как правило, невелики, поэтому узким местом являются именно дисковые операции.

Для задач молекулярной биологии, эта проблема также актуальна, однако здесь есть и своя специфика. В последнее время увеличивается количество баз данных, содержащих информацию о цепочках ДНК и РНК различных видов живых организмов. Поиск в таких базах данных осуществляется для нахождения похожих участков хромосом с целью исследования эволюционных процессов. Как правило, представление ДНК в виде строки, где буквы соответствуют различным нуклеотидам, вместе с расстоянием редактирования в качестве меры близости, является адекватной моделью для сравнения участков хромосом. Основная сложность заключается в том, что разные типы ошибок имеют разную стоимость. Это сильно затрудняет оптимизацию. Точнее говоря, на сегодняшний день она производится за счет полноты поиска: у наиболее часто используемых алгоритмов BLAST и FASTA ([8]) процент "успеха" нахождения гомологичных цепочек составляет примерно 80%.

Что касается задач сравнения мультимедиа данных или графов с большим количеством вершин и ребер, то здесь время, затрачиваемое на сравнение элементов, может значительно превышать время считывания с диска. Так, например, в случае графов, не найдено алгоритма полиномиальной сложности, проверяющего изоморфность двух графов: для известных алгоритмов с увеличением числа вершин, время сравнения растет экспоненциально.

Для случая "дорогих" функций похожести существуют специальные алгоритмы (см. далее на стр. 42), позволяющие на один-два порядка в зависимости от количества ошибок уменьшить число операций сравнения и считывания элементов с диска. К сожалению, как я уже отмечал, в случае "небольших" элементов таких, как ключевые слова, операция сравнения является исключительно "дешевой" и время ее выполнения составляет несколько процентов от времени дисковых операций. Более того, в силу малых размеров элемента по сравнению с размером страницы вместе с самим элементом необходимо считывать множество "соседей", которые не являются потенциальными кандидатами на неточное соответствие. Другими словами, произвольный дисковый доступ к элементам малого размера — непозволительная роскошь. Это негативное явление усугубляется с увеличением числа ошибок, допустимых в терминах запроса, когда количество "почти" совпадающих, разбросанных по разным дисковым страницам, элементов резко возрастает. В силу вышеприведенных причин очень сложно добиться превосходства над программой, сканирующей индекс, даже при небольшом количестве ошибок.

Информационный поиск (ИП) в документальных базах данных является наиболее актуальной и исследованной областью, в которой применяются поисковые алгоритмы, в том числе, обеспечивающие поиск на неточное соответствие. Эта проблема имеет богатую историю и ее значимость трудно переоценить, поэтому я начну свой обзор поисковых методов с описания структур данных ИПС. В значительной степени я буду следовать обзору Райзенбергена ([26]).

Несмотря на то, что данная работа была написана еще в 1979 году, это один из первых и наиболее полных обзоров по информационному поиску (ИП) известного специалиста в этой области. В ней Райзенберген описал основные

Таблица 1:

	ВД	ИП
Соответствие	точное	частичное соответствие, наилучшее соответствие
Вывод	дедуктивный	индуктивный
Классификация	детерминированная	вероятностная
Язык запросов	искусственный	естественный
Полнота запроса	“точный” запрос	присутствует неопределенность
Критерий выборки документов	булева функция соответствия	функция релевантности
Устойчивость к ошибкам	неустойчива	устойчивый

структуры данных и стратегии поиска в информационно-поисковых системах (ИПС), а также предсказал многие тенденции развития систем ИП.

1.4.1 Структуры данных систем информационного поиска (ИП)

ИП слишком широко используемый термин, поэтому я ограничусь рассмотрением автоматических информационно-поисковых текстовых систем (ИПС). Информация также является слишком широко определенным термином: в контексте ИП удобно считать информацией набор текстовых документов. ИП, впрочем, как и выборка данных (ВД), в отличие от экспертных систем, не создает новое знание (т.е. не осуществляет вывод новых фактов, используя уже существующие базы знаний), а лишь проверяет наличие в информационном массиве документов релевантных запросу. Между ИП и ВД не существует четкой границы. Эти области имеют много общих аспектов, как концептуальных, так и алгоритмических. Тем не менее, можно выделить набор признаков, позволяющих отличить ИПС от систем ВД, которые приведены в таблице 1.

Рассмотрим эти признаки подробнее. Системы ВД обычно осуществляют поиск на точное соответствие, проверяя наличие или отсутствие определенного элемента данных, как правило ключевого слова, в документе. В процессе ИП могут быть найдены ключевые слова, частично совпадающие со словами поискового запроса, и выбраны наиболее похожие.

Вывод, лежащий в основе классификации, в системах ВД имеет простую дедуктивную основу: если Документ1 соответствует Документу2, а Документ3 – Документу2, то Документ1 соответствует Документу3. Другими словами, в основе классификации лежит отношение эквивалентности. В ИПС ассоциация может задаваться с определенной степенью уверенности. В основе вывода в ИПС лежит вероятностный подход, а в основе систем ВД — детерминированный.

Опять-таки, классификация может быть основана на наличии у документа

определенных признаков, необходимых и достаточных для принадлежности классу, либо для определенных документов отношение "у документа есть атрибут ..." может быть задано с некоторой точностью, или, что тоже самое, рандомизировано.

Язык запросов в системах ВД искуственен, а условие поиска задается булевой функцией, построенной на основе предиката **содержит ключевое слово X** и связок OR, AND и NOT. В данном случае функция релевантности принимает только два значения, и ключевые слова запроса должны совпадать со словами документа. В ИПС для запроса можно использовать фразы естественного языка. Разумеется, при этом возникает неопределенность задания поискового условия в отличие от систем ВД, а документы выдаются в порядке убывания значения релевантности. С другой стороны, ошибки в поисковом запросе для систем ВД приводят к выборке совершенно ненужных документов, тогда как ИПС существенно более устойчивы к таким неточностям.

Чтобы организовать поиск в информационном массиве документов, нужно представить документы в электронном виде. При этом документ может быть представлен в виде полнотекстовой версии, заголовка, резюме или списка слов. Также необходимо произвести автоматический анализ текстов. Ввиду несовершенства методов, сложности и дороговизны реализации семантические и лингвистические методы анализа документов не получили широкого распространения. С другой стороны, статистический подход используется еще с давних времен, и проверен на практике.

Статистический подход был намечен в работах Луна ([18]), который предположил, что частота появления слова является подходящим критерием оценки его значимости. Лун одним из первых использовал данную характеристику для вычисления значимости отдельных фраз и документов. При этом он учитывал следующую полученную экспериментальным путем закономерность (Закон Ципфа): наиболее значимые слова находятся "посередине", то есть не входят в число ни наиболее редко, ни наиболее часто встречающихся слов. Несмотря на свою относительную простоту, идеи Луна лежат в основе большинства современных ИПС.

Процесс автоматического анализа текста, как правило, включает в себя следующие стадии:

- исключение часто и редко встречающихся слов.
- отсечение суффиксов (выделение словоформ).
- определение эквивалентных основ.

В процессе классификации основы, как правило, не разделяются на классы "жестко": для каждой пары терминов может быть установлено некоторое корреляционное значение, определяющее степень взаимозаменяемости слов. Это значение может быть получено, как в результате обработки текстов человеком, так и в результате автоматической классификации (см., например, [7]).

Таблица 2:

w_1	$\rightarrow$	R_1, R_2
w_2	$\rightarrow$	R_2, R_3, R_4
w_3	$\rightarrow$	R_5

Центральным моментом систем текстового поиска являются структуры данных. В структуре данных любой информационной системы можно выделить логический и физический уровень. ИПС — не исключение.

В процессе разработки физической модели данных внимание уделяется, в основном, оптимальности распределения дискового пространства и скорости выборки, тогда как в случае логической — взаимосвязи различных компонентов. В основном я буду рассматривать аспекты именно логической структуры данных, затрагивая аспекты физической организации ИПС только там, где они тесно переплетаются с логическими.

1. Каталоги или простые индексы. При рассмотрении моделей ИПС я буду часто использовать понятие **представления документа**. В самом общем случае, под представлением документа понимается список слов (с учетом порядка их следования), составляющих этот документ[6]. Однако, как правило, целесообразно бывает использовать более "компактное" представление.

Например, в уже упомянутой работе Райзенбергена используется следующее понятие представления. Пусть документ D содержит ключевые слова $w_1, w_2, \ldots, w_n$ с частотами вхождения $p_1, p_2, \ldots, p_n$. Будем называть, n пар вида $R = \{(p_1, w_1)(p_2, w_2) \ \ldots (w_n, p_n)\}$ представлением документа или, по другому, его записью.

Каждая запись документа состоит из полей, содержимым поля является пара: (**значение частоты вхождения, ключевое слово**). Кроме того, у каждого поля записи есть указатель, задающий физическое расположение этого поля. Зная указатель, можно всегда считать значение поля. В ИПС записи объединяются в логические группы, называемые файлами. Указатели позволяют объединять поля (записи) в списки. Будем считать, что все записи, содержащие одинаковые ключевые слова, могут быть соединены в списки (не обязательно в один), причем так, что зная указатели на начала списков $A_i(w)$ можно прочитать все записи, содержащие ключевое слово w. В таблице 2 проиллюстрировано использование списков записей.

Множество таких списков называют каталогом, простым индексом, или также **инвертированным файлом (ИФ)**. Такой список является инвертированным в том смысле, что атомарные атрибуты документа, слова, извлекаются из самих документов на поверхность, и создаются новые документы, содержащие списки вхождений слов в исходные документы, которые также называют

[6]Разумеется, что в данном случае делается допущение, что документ всегда можно однозначно и осмысленно разбить на слова.

инвертированными списками.

Если нужно инвертировать текст целиком, то используется полное представление документа, в котором для каждого слова хранится информация о контексте, например, его порядковый номер в документе или адрес. В этом случае инвертирование является полным и размер индекса может быть очень велик. В некоторых случаях его размер может превышать размер исходного массива документа в несколько раз. В приведенном выше примере из [26] инвертирование является частичным. Хочу отметить, что использование такого представления документа получило наибольшее распространение, а данные по количеству вхождений ключевых слов в документ получили название **статистики**.

Большие информационные массивы обладают большими индексами, поэтому в реальных ИПС для ускорения доступа данные организованы в виде иерархии индексов. Райзенберген производит детальный анализ различных типов каталогов (индексов) и подразделяет их на **последовательные файлы**, **инвертированные файлы (ИФ)**[7], **индексно-последовательные файлы**, **мульти-списки**, и **блочные мульти-списки** (см. подробнее [26]). В настоящее время из всего этого многообразия используются, в основном, многоуровневые индексно-последовательные файлы, или, по другому, инвертированные списки со словарем. Индекс в данном случае действительно многоуровневый, потому что для эффективного доступа, как списки вхождений, так и словарь ключевых слов должны быть специальным образом проиндексированы.

Помимо, стандартных логических структур таких, как ИФ, на множестве записей документов могут быть введены дополнительные структуры, задающие классификацию документов. Прежде всего, документы могут быть разбиты на категории с помощью древовидной структуры по смысловым признакам. Документы, располагающиеся в определенном поддереве считаются принадлежащими одной категории, при этом для каждой категории-узла дерева K может быть более детальное разделение на подкатегории, $K_1, K_2, \ldots, K_n$, которые являются дочерними узлами K. В общем случае документ может быть ассоциирован с несколькими категориями и у него могут быть ссылки на другие узлы. При этом дерево преобразуется в направленный граф, в котором у одной вершины может быть несколько родителей.

Райзенберген отмечает, что задание классификации на множестве документов вторично, по сравнению с выбором файловой структуры, тем не менее, многие авторы выделяют файлы, расширенные категоризующей структурой, в отдельную группу. Возьмем, например, кластеризованные файлы (которые подробнее описаны ниже), которые Салтон и многие современные авторы не без оснований считают самостоятельной логической моделью ИПС, хотя по мнению Райзенбергена кластеризация — это всего лишь метод автоматической классификации.

2. Сигнатурные файлы (СФ). Долгое время считалось, что ИФ обречены "съедать" значительное дисковое пространство. Поэтому в качестве

[7]Райзенберген использует термин ИФ только для частного случая простого индекса

альтернативы ИФ для поиска по ключевым словам были придуманы сигнатурные файлы (СФ). Следуя статье [17], я приведу описание СФ и результаты сравнения с ИФ по нескольким параметрам. Авторы [17] сравнивали производительность, объем индекса и функциональность системы Managing Gygabyte [8], с различными реализациями СФ. В итоге авторы пришли к выводу, что ИФ почти по всем параметрам значительно превосходят СФ. В качестве исходных информационных массивов использовались статьи Wall Street Journal, а также случайный массив документов, набор слов которого и вероятности их появления были получены в результате обработки большого массива реальных документов в системе Интернет, а количество слов в каждом документе задавалось случайной функцией с гамма распределением, функция распределения которого выражается формулой:

$$F(t) = \frac{t^{\alpha-1}e^{-t/\beta}}{\beta^{\alpha}(\alpha-1)!}$$

В качестве СФ использовались так называемые битовые срезки (БС) и блочные битовые срезки (ББС). В самой простой реализации СФ, битовых строках, для каждой записи строится сигнатура фиксированной длины — битовый массив (вектор) длины m. Для каждого ключевого слова w с помощью хэш-функции f определяется номер элемента $f(w)$, соответствующий этому слову, и этому элементу присваивается значение 1. Таким образом, в сигнатуре документа $i-$ый элемент равен 1 тогда и только тогда, когда $i = f(w)$, для какого-либо термина документа w. В процессе поиска по запросу: **документ содержит ключевое слово** v считываются сигнатуры всех документов и проверяется равен ли единице бит с номером $f(v)$. Если да, то соответствующий документ сканируется на предмет наличия в нем необходимого слова.

Основным недостатком данного метода является необходимость в считывании всего списка сигнатур. Чтобы преодолеть эту трудность применяют, метод битовых срезок (БС). При этом сигнатура битовых строк "разрезается" на части, для каждого целого i от 1 до m (длина сигнатуры) создается битовый массив, длина которого равна количеству документов N, а единичные элементы соответствуют номерам документов, содержащих хотя бы одно ключевое слово из множества $f^{-1}(i)$, то есть все слова, которые хэш-функция отображает в целое i. Стратегия поиска следует непосредственно из построения индекса: для всех ключевых слов w запроса вычисляются значения $f(w)$ и выбираются соответствующие битовые срезки, которые затем пересекаются (или соединяются с помощью дизъюнкции, если в поисковом запросе содержатся связки OR) для определения документов, которые могли бы содержать ключевые слова запроса. К сожалению, если документов много, например 1млн., то размер битовой срезки может достигать одного мегабита. Чтобы уменьшить размер битовой срезки применяют метод блочной битовой срезки (ББС), в котором $k-$ый бит срезки соответствует не одному документу, а целому блоку, количество элементов которого определяется заранее.

[8] Описание системы Mg и ссылка на исходные тексты может быть найдена по адресу: ftp://munnari.oz.au/pub/mg.

Как на реальном наборе данных, так и на созданном случайным образом, экспериментальные результаты отличаются весьма незначительно, и как битовые срезки (БС), так и блочные битовые срезки (ББС) проигрывают упакованным ИФ почти по всем параметрам, в том числе и по времени поиска. В частности:

- Сжатые ИФ занимают гораздо меньше места на диске. Так в [17] индекс ИФ составлял 5-6%% от общего объема документов, тогда как у СФ его размер доходил до 30-40%% объема исходного информационного массива.

- ИФ гораздо лучше сжимаются. Для того, чтобы поиск в СФ был эффективен, вероятность появления единичного бита в одном байте должна быть равна примерно одной восьмой. При этом энтропия распределения Бернули равна:

$$-\sum_{k=0}^{n} C_n^k p^k q^{n-k} \log_2(p^k q^{n-k}) = -\sum_{k=0}^{n} C_n^k p^k q^{n-k}(k \log_2 p + (n-k)\log_2 q) =$$

$$= -\sum_{k=1}^{n} nC_{n-1}^{k-1} p^k q^{n-k} \log_2 p - \sum_{k=0}^{n-1} nC_{n-1}^{k} p^k q^{n-k} \log_2 q =$$

$$= -np \log_2 p \sum_{k=0}^{n-1} C_{n-1}^k p^k q^{n-1-k} - nq \log_2 q \sum_{k=0}^{n-1} C_{n-1}^k p^k q^{n-1-k},$$

  где n — длина кодируемого слова в битах, p — вероятность появления единичного бита в слове, $q = 1 - p$. Поскольку обе суммы в выражении для энтропии равны единице получаем, что энтропия равна:

$$n(p \log_2(1/p) + q \log_2(1/q)).$$

  Если вероятность появления единичного бита равна $p = 1/8$, (то есть примерно один бит в байте[9]), то энтропия составляет приблизительно $0.53n$. Согласно теореме Шенона — это минимальная средняя длина текста, сжатого энтропийными методами такими, как метод Хаффмана или арифметического кодирования. То есть коэффициент сжатия равен примерно двум. Индекс ИФ сжимается значительно лучше: исходя собственного опыта, я могу сказать, что ИФ вполне можно сжать в 4-6 раз.

- СФ очень чувствительны к выбору параметров и неудачный их выбор сильно сказывается на производительности и эффективности поиска. Это неприятная особенность отмечена многими исследователями, в том числе об этом упоминается и в [37]. В отличие от СФ индексирование ИФ с помощью B^+ деревьев с большим коэффициентом ветвления подходит для абсолютного большинства документальных баз данных и не требует дополнительной настройки.

[9] Степень разреженности сигнатуры выбирается из практических соображений. Для оптимального по времени поиска и размеру индекса СФ примерно один бит в байте должен равняться единице, а остальные нулю.

Таблица 3: Время булевого поиска в ИФ и СФ

	ИФ	БС	ББС
First	0.09	0.35	0.16
Second	0.21	0.38	0.28
Third	1.33	3.76	4.22
Fourth	0.28	0.46	0.24
Fifth	1.12	0.83	0.54
Queens	0.50	2.50	1.94

Таблица 4: Время ранжирования документов в ИФ и СФ

	ИФ	БС	ББС
First	0.49	1.24	0.52
Second	1.16	4.11	5.69
Third	2.52	12.26	10.80
Fourth	0.33	0.85	0.37
Fifth	1.68	3.62	4.14
Queens	5.21	18.60	16.28

- Если словарь ИФ хранится в виде дерева с больших количеством листьев у внутренних вершин, то внутренние вершины занимают мало места и, как правило, могут быть загружены в оперативную память, что существенно ускоряет поиск.

- Упакованный индекс ИФ занимает всего 6-10%% от исходного объема, по сравнению с 30 процентами индекса СФ.

- Время как булевого, так и ранжированного поиска (см. ниже описание стратегий поиска) в ИФ меньше, чем время, соответственно, булевого и ранжированного поиска в СФ. Данные по среднему времени поиска в СФ и ИФ, полученные в [17], приведены в таблицах 3 и 4. В экспериментах использовались пять случайным образом созданных наборов булевых поисковых запросов и один реальный[10], и реализации двух различных типов СФ: битовые срезки и блочные битовые срезки.

  Приведенные данные показывают, что поиск в СФ происходит не быстрее, а в большинстве случаев намного медленнее, чем в ИФ.

- ИФ создается быстрее, чем СФ (в два раза по данным [17]).

[10]Набор типичных запросов к базе материалов TREC.

3. Векторные модели (ВМ) и кластеризация. Идея, лежащая в основе ВМ, заключается в следующем. Разбиваем документы на элементарные термины t_i, например, на ключевые слова, или три-граммы[11] и сохраняем их в специальном словаре, попутно нумеруя. Предположим, что всего в документальной БД есть n элементарных терминов. Тогда каждый документ D_i, содержащий термины с номерами $t_1^i t_2^i \dots t_{k(i)}^i$, можно отобразить в вектор размерности n, сигнатуру документа, у которого только компоненты с номерами $t_1^i t_2^i \dots t_{k(i)}^i$ отличны от нуля и равны $w_1^i w_2^i \dots w_{k(i)}^i$. Значения w_j^i будем называть весами термина, а вектор весов для краткости я буду иногда обозначать: $w_i = \left(w_1^i, w_2^i, \dots, w_{k(i)}^i\right)$ как вектор размерности $k(i)$, пропуская в записи нулевые компоненты. Аналогичный вектор весов можно составить и для поискового запроса.

Выбор значений w_j^i (взвешивание терминов) может осуществляться различными способами. Вес термина (см. [4] гл. 12.8.2) в документе можно получить, например, следующим образом. Пусть f_r^i есть число вхождений термина r в документ D_i. Общее количество вхождений термина r во все документы обозначим через F_r. $F_r = \sum_{i=1}^m f_r^i$, где m равно числу документов. Вес термина r в документе i положим равным $w_r^i = f_r^i / F_r$.

Для определения степени близости $Corr(D_i, D_j)$ применяются различные меры сходства (релевантности), которые также могут использоваться и для ранжирования результатов выборки при поиске в ИФ и СФ. Наиболее известными являются ([4] гл. 12.7.3):

Коэффициент Дайса

$$Corr(D_i, D_j) = \frac{2\left(\sum_{k=1}^n w_k^i \cdot w_k^j\right)}{\sum_{k=1}^n w_k^i + \sum_{k=1}^n w_k^j} .$$

Коэффициент Жаккара

$$Corr(D_i, D_j) = \frac{\sum_{k=1}^n w_k^i \cdot w_k^j}{\sum_{k=1}^n w_k^i + \sum_{k=1}^n w_k^j - \sum_{k=1}^n w_k^i \cdot w_k^j} .$$

Коэффициент косинуса

$$Corr(D_i, D_j) = \frac{\sum_{k=1}^n w_k^i \cdot w_k^j}{\sqrt{\left(\sum_{k=1}^n {w_k^i}^2\right)\left(\sum_{k=1}^n {w_k^j}^2\right)}} .$$

В качестве стратегии поиска можно использовать последовательный перебор: выбрать пороговое значение ϵ, вычислить корреляционное значение $Corr(D_i, Q)$, где Q — представление поискового запроса, и выдать пользователю только те документы, у которых величина $Corr(D_i, D_q)$ превышает ϵ.

Последовательные поиски работают достаточно медленно, потому что требуют вычисления корреляционного значения для всех документов и, соответственно, считывания всех сигнатур.

[11] То есть подстроки длиною в три символа, по-другому, триады.

Чтобы преодолеть этот недостаток, похожие документы группируются в кластеры, и на первом этапе определяются релевантные кластеры, подлежащие более детальному анализу. Кластерный поиск основан на гипотезе, что релевантные документы имеют тенденцию быть релевантными одним и тем же запросам. "Кластеризация документов ... порождает однородные группы документов, обладающие тем свойством, что документы внутри такой группы более тесно связаны друг с другом, чем с документами из других групп" ([4] гл. 12.9 стр. 567).

Необходимо сказать несколько слов о методах представления кластеров. Для этих целей строится специальный документ: так называемый профиль или центроид. Ожидается, что центроид позволяет судить есть ли в кластере документы, релевантные запросу.

В качестве представителя кластера можно выбрать, например, наиболее "типичный" документ. Однако среди документов может не оказаться достаточно характерного представителя, поэтому представителя часто просто вычисляют.

Если $\{D_1, D_2, \ldots, D_p\}$ — множество документов кластера, где каждый представлен числовым вектором весов $(w_1^i, w_2^i, \ldots, w_{k(i)}^i)$, то в качестве центроида можно выбрать вектор, вычисляемый, к примеру, по формуле:

$$C = \frac{1}{p} \sum_{1}^{p} \frac{w_i}{||w_i||},$$

где $||.||$ – евклидова норма. Построение центроида — сложная задача. При ее решении нужно учитывать множество факторов:

- Устойчивость кластеров относительно модификаций документов и ошибок в их описании.
- Размер областей пересечения кластеров.
- Эффективность и полноту выборки.

Следующие два небольшие подраздела посвящены основным типам одноуровневых индексов.

4. **Деревья.** Как только появляется набор записей, который можно идентифицировать по некоторому ключу, например, набору ключевых слов, встает проблема эффективной организации поиска и обновления этого набора. Самый простой вариант: организовать записи в виде списка. Он же один из самых неэффективных, потому что при поиске количество операций имеет порядок $\mathcal{O}(N)$, где N — количество записей в наборе. Чтобы улучшить поисковые характеристики, можно хранить записи в отсортированном виде и производить дихотомический поиск. Давно было замечено, что поиск в отсортированном массиве, требует логарифмического количества операций. К сожалению, количество операций, необходимых для добавления записи имеет порядок $\mathcal{O}(N)$.

Структура данных, которая позволяет осуществлять как поиск, так и вставку за $\mathcal{O}(\log_2 N)$ операций — бинарное (в общем случаем количество вершин может быть больше двух) сбалансированное дерево. В бинарном

Таблица 5:

по	$\longrightarrow$	л $\rightarrow$ е
$\downarrow$		$\downarrow$
рог		о $\rightarrow$ г
		$\downarrow$
		жение

дереве в каждом узле хранится ключевое значение K и ссылки на запись и поддеревья (если узел нетерминальный), при этом все ссылки на записи с ключами меньшими либо равными K находятся в левом поддереве, а с ключами большими K — в правом. Если дерево, сбалансировано, то есть у каждого нетерминального узла, за исключением быть может одного узла предпоследнего уровня, два дочерних узла, то количество поисковых операций составляет примерно $\log_2 N$, где N — количество записей в дереве. Количество операций, необходимых для занесения новой записи, имеет тот же порядок. К счастью, существуют алгоритмы, позволяющие при вставке сохранять сбалансированность дерева.

Кроме сбалансированных n-арных деревьев, в последнее время для хранения текстовой информации все чаще используются trie-деревья. Основная идея, заключается в том, чтобы разместить все записи с одинаковым префиксом в одном поддереве. Схема trie-дерева для множества слов *поле*, *полог*, *порог* и *положение* изображена в таблице 5.

Одним из достоинств trie-деревьев является компактное представление данных. При использовании подходящих алгоритмов можно добиться 30-50 процентного сжатия словаря. Trie-деревья хорошо подходят для хранения морфологической информации в словарях сильно флективных языков таких, как русский. Различные модификации trie-деревьев часто являются основой для построения суффиксных деревьев (см. подробнее [16]).

Другим известным методом организации данных является хеширование.

5. Хеширование. Предположим, что на множестве ключей определена функция $f(K)$, отображающая это множество в конечный отрезок целых чисел, например от 1 до 10000. В дальнейшем будем называть ее хэш-функцией. Зарезервируем массив ключей размером в 10000 элементов, и при добавлении нового ключа K будем помещать его в ячейку с номером $f(K)$. Если $f(K_1) \neq f(K_2)$ для всех пар ключей, то как для поиска, так и добавления элемента с ключом K достаточно одного обращения к элементу массива с номером $f(K)$. Поскольку построение таких функций f — сложная задача, на практике допускаются коллизии, а разные K_1 и K_2 могут отображаться f в одно и то же число. Существует два основных способа разрешения коллизий.

1. При добавлении нового элемента помещать его в ближайшую свободную ячейку, или ближайшую ячейку с номером, вычисляемым по определенному правилу.

2. Хранить в ячейке ссылку на список ключей с одинаковыми значениями хэш-функции.

Преимущество хеширования состоит в простоте реализации и высокой скорости поиска. Недостатком является возможность ускорения поиска только на точное равенство, тогда как деревья позволяют ускорить также выборку диапазона ключей, то есть всех ключей K, лежащих в промежутке от K_1 до K_2. На этом я закончу рассмотрение методов индексирования на основе деревьев и хэш-таблиц, а интересующиеся могут прочитать подробно о них в третьем томе бестселлера Д. Кнута ([3]), который буквально недавно был переиздан в России.

1.4.2 Стратегии поиска

Рассмотрев основные методы организации данных, перейдем к рассмотрению различных поисковых стратегий. Все поисковые стратегии построены на сравнении поискового запроса и документа. Часто это сравнение бывает неявным, как, например, в кластерных файлах: в данном случае поисковый запрос сравнивается с представителем кластера. Большинство поисковых стратегий можно разделить на три большие группы: **булевый поиск**, **ранжированные поиск** и **кластерный поиск**.

1. Булевый поиск.

Булевый или логический поиск использует функцию релевантности, принимающая только два значения: 0 и 1. Эта формулировка имеет смысл только в том случае, когда запрос представлен в виде булевой функции, содержащей предикаты $Exists(w)$ — содержит ключевое слово w, а также связки AND, OR и NOT. В языках запросов, используемых в большинстве ПС, вместо предиката $Exists('\text{ключевое слово}')$ используют просто 'ключевое слово'. Так например, запрос: **машина AND автомат AND NOT двигатель** вернет документы, которые содержат ключевые слова **машина** и **автомат**, но не содержат слово **двигатель**. Некоторые поисковые системы позволяют пользователю расширять запрос, используя словарь синонимов. Так например, запрос на поиск всех документов, в которых упоминаются машины может выглядеть следующим образом: **синоним(машина)**.

Считается, что ИФ и СФ являются наиболее подходящим методами организации данных для реализации булевого поиска.

Естественным расширением булевого поиска является ранжирование найденных документов по степени релевантности, или ранжированный поиск.

2. Ранжированный поиск. Если ПС умеет вычислять функцию соответствия $Corr(D_i, Q)$ между запросом Q и документом D_i, то ее значения можно использовать для сортировки найденных документов. Таким образом, у пользователя появляется возможность просмотреть сначала те документы, которые с большей степенью достоверности содержат нужную информацию. В случае

ранжированного поиска ПС сначала осуществляет булевый поиск по ключевым словам запроса, их словоформам и синонимам. При этом ПС находит документы, содержащие хотя бы один термин. Затем полученные результаты ранжируются и выдаются пользователю. Типичным представителем ИПС, осуществляющей ранжированный поиск является разрабатываемая IBM система WEB-поиска Clever, которая "... начинает со стандартного поиска по ключевым словам, позволяющего получить базовое множество страниц. Затем ведется поиск документов, которые имеют ссылки на эти страницы или на которые ссылаются страницы из базового множества. Clever классифицирует страницы базового множества и связанные с ними страницы по количеству ссылок на них." ([1]). Подобным образом поступают также такие известные поисковики, как Яндекс и Yahoo.

3. Векторные модели и кластерный поиск. Как уже говорилось выше ключевым элементом ПС, использующих векторные модели и кластерные методы, является функция похожести или соответствия документов.

Пусть $|D|$ обозначает размер множества ключевых слов документа, Q — поисковый запрос, тогда

$$Corr(D,Q) = \frac{2|D \cap Q|}{|D| + |Q|}$$

еще один пример функции соответствия.

В системе SMART ([27]) запрос и документ представлены вектором большой размерности (сигнатурой), i-ый элемент которого равен весу i-го ключевого слова: $sign(D) = (w_1, w_2, \dots w_n)$, $sign(Q) = (q_1, q_2, \dots q_n)$. При этом функция корреляции равна косинусу угла между сигнатурами:

$$Corr(D,Q) = \frac{\sum\limits_{i=1}^{n} w_i q_i}{\left(\sum\limits_{i=1}^{n} w_i^2\right)^{1/2} \left(\sum\limits_{i=1}^{n} q_i^2\right)^{1/2}}$$

Самый простейший вариант стратегии поиска с использованием функции соответствия — последовательный поиск.

3.1 Последовательный поиск. В общих чертах, он уже описан в разделе, посвященному кластерным моделям ИПС. Пусть в базе есть n документов и выбрано пороговое значение ϵ. Тогда последовательно считав документы, вычислив значения $r_i = Corr(D_i, Q) = Corr(sign(D_i), sign(Q))$, отобрав только те документы, для которых r_i превышает порог, и отсортировав найденные документы по убыванию величины r_i, получаем результат поиска.

Одна из проблем заключается в выборе подходящего порогового значения, поскольку заранее неизвестно, какое значение для конкретного запроса является наилучшим.

Особенно это актуально для поиска по сходству с построением сигнатуры на основе частотных характеристик n-грамм.

Рассмотрим пример слов **порог** и **подог**, в котором **подог** получается из **порог** в результате опечатки. Множества три-грамм (или триад) данных слов (т.е. всевозможных подстрок длины три) приведены в таблице 6.

Таблица 6:

порог	пор	оро	рог
подог	под	одо	дог

Пронумеруем три-граммы в порядке их следования в таблице: слева направо и сверху вниз. Среди три-грамм двух близких слов нет ни одной общей, а это значит, что, функция корреляции, вычисляемая как косинус угла между векторами с единицами в i-ом разряде, если i-ая три-грамма есть в слове, и 0 в противном случае, будет равна -1, а это минимально возможное значение!

Многие известные методы ([12]) используют в качестве компонент сигнатуры значения частот вхождения соответствующих n-грамм минус некоторые средние величины. Если использовать формулу вычисления вектора сигнатур, реализованную в системе TELLTALE ([29]):

$$d_i = c_i/m - a_i,$$

где d_i — i-ый элемент вектора сигнатуры, c_i — количество n-грамм с номером i в слове (документе), m — общее количество n-грамм в слове, и a_i — средняя частота вхождения i-ой n-граммы в словарь, получаем следующие выражения для сигнатур:

$$sign(\text{порог}) = (1/3 - a_1, 1/3 - a_2, 1/3 - a3, -a_4, -a_5, -a_6)$$

$$sign(\text{подог}) = (-a_1, -a_2, -a_3, 1/3 - a_4, 1/3 - a_5, 1/3 - a_6)$$

Если в качестве функции корреляции выбран коэффициент косинуса то $Corr(\text{порог}, \text{подог})$ равна:

$$\frac{-(a_1 + a_2 + a_3 + a_4 + a_5 + a_6)/3 + a_1^2 + a_2^2 + a_3^2 + a_4^2 + a_5^2 + a_6^2}{|sign(\text{порог})||sign(\text{подог})|}.$$

Поскольку количество три-грамм обычно велико, то маловероятно, чтобы частота вхождения в словарь какой-либо три-граммы была больше $1/3$. А это, в свою очередь, означает, что функция корреляции, скорее всего, будет не больше нуля. С другой стороны, рассматриваемые слова отличаются только на одну букву.

3.2 Кластерный поиск.

Последовательный поиск не слишком эффективен, потому что требует вычисления корреляционного значения для всех документов и, соответственно, считывания всех сигнатур. Для сокращения перебора применяют методы объединения документов в кластеры, описанные выше, и в процессе поиска сравнивают запрос сначала только с представителями кластеров. Поиск внутри кластера осуществляется только если значение функции $Corr(C_i, Q) > \epsilon$, где C_i — представитель кластера (центроид), а ϵ — пороговое значение.

Подводя итоги, можно выделить две принципиально различные модели данных:

1. Инвертированные либо сигнатурные файлы с разбиением документов на слова.

2. Векторные модели (ВМ).

Хотя Райзенберген, впрочем как и многие другие авторы, не касается вопросов неточного равенства, как ИФ, так и ВМ могут быть использованы для поиска на неточное равенство. Для ИФ необходимо модифицировать поисковые алгоритмы словаря, а в ВМ использовать сигнатуры на основе, например, n-грамм. Как я уже отмечал (см стр. 24), корреляционные функции на основе n-грамм не всегда адекватно оценивают величину ошибки, потому что изменение одной буквы приводит к изменению n n-грамм. Несмотря на это, например, изменение трех три-грамм, при условии, что поисковый запрос содержит несколько слов и, соответственно, 10-20 три-грамм, не сильно повлияет на значение функции корреляции. В таких случаях также помогает введение так называемых "синтетических" n-грам, то есть префиксов терминов длины меньшей n. Как правило, вероятность допустить ошибку в начале слова несколько меньше, чем, да и более короткие n-граммы к тому же имеют часто больший вес, что несколько компенсирует ошибку в середине слова, которая приводит к модификации сразу множества подстрок слова. Некоторые варианты реализации сигнатурных файлов (СФ), такие, как блочные битовые срезки (см. выше описание СФ), использующие словарь также могут быть адаптированы для решения этой задачи.

Я хочу еще раз подчеркнуть, что существует очень немного коммерческих и свободно распространяемых поисковых систем, реализующих поиск на неточное равенство. Хотя полнотекстовые индексы, с возможностью поиска по ключевым словам и их основам встроены в такие системы, как SYBASE и Lotus Notes, единственной широко распространенной поисковой системой, реализующей данную возможность является glimpse [12] ([37]).

Glimpse также создает словарь ключевых слов но без стемминга, то есть, не производя усечение до основ. ПС glimpse позволяет осуществлять поиск по подстроке и различным регулярным выражениям, допускающий ошибки в задании запроса. Glimpse использует следующую модификацию метода битовых срезок: он хранит только список номеров блоков, куда входит документ, то есть только "верхушку" индекса СФ. Таким образом, все множество документов разбивается на несколько блоков, и в словаре хранится список вхождений ключевого слова в каждый из блоков.

Основным достоинством, по мнению авторов, является небольшой размер индекса: около 2-4%% от общего объема, и большие возможности для поиска по шаблонам различных типов. Недостатком же является низкая скорость поиска в информационном массиве, содержащем много маленьких файлов, потому что эти файлы, как правило, оказываются разбросанными по диску, а скорость произвольного доступа очень невелика. Чтобы преодолеть эту трудность, необходимо "перестраивать" набор файлов, например, объединяя маленькие

[12] а также webglimpse, которые не являются коммерческими системами

файлы в файлы большего размера. При этом чтение каждого из файлов будет происходить с пиковой скоростью, составляющей нескольких мегабайт в секунду.

Glimpse неплохо подходит для использования на персональном компьютере, потому что в пользователю важнее сэкономить дисковое пространство, и получить дополнительные поисковые функции, чем выиграть в скорости, скажем в 5-20 раз.

Следует отметить, что создатели glimpse'а считают, что индекс ИФ, занимает очень много места: 50-300%% исходного объема. Однако, если учесть, что для системы Mg, описанной в [17], индекс ИФ, составляет всего несколько процентов общего объема, этот аргумент кажется сомнительным. При добавлении в индекс новых записей, объем индекса в процентном отношении возрастает из-за потерь при расщеплении страниц, но даже в случае динамически изменяемой совокупности данных процент размера индекса от общего объема документов не превышает 14.

1.4.3 Алгоритмы вычисления функции расстояния

Чтобы осуществлять эффективный поиск на неточное равенство, нужно уметь эффективно сравнивать строки "неточно". Для этого, прежде всего, нужно построить модель сравнения строк, то есть определить, что же понимается под неточным равенством. Вообще говоря, существует множество способов введения функции расстояния. Например, в качестве функции расстояния можно взять метрику Левенштайна, которая определяется как минимальное количество модификаций, преобразующих строку u в v. Однако я, следуя [6], начну с рассмотрения более общей функции расстояния, частным случаем которой является метрика Левенштайна $edist_1$, и алгоритма ее вычисления с использованием метода динамического программирования.

Расширим алфавит $\mathcal{A}$, расширенный пустым символом ϵ. Введение пустого символа позволяет свести все операции редактирования (за исключением транспозиции символов) к операции замены.

Определение 1.9 *Выравниванием слов* $u = u_1u_2 \dots u_m$ *и* $v = v_1v_2 \dots v_n$ *длины* l *будем называть пару строк* u' *и* v' *длины* $l \geq \max(m, n)$*, где слова* u' *и* v' *получаются из исходных слов* u *и* v *в результате добавления пустых символов* ϵ *таким образом, чтобы в обоих словах* u' *и* v' *не было пустых символов в одинаковых позициях.*

Пример. $u =$ **поле,** $v =$ **полог. Тогда пара строк:** $u' =$**поле**ϵ**,** $v' =$**полог, является примером выравнивания.**

В случае метрики Левенштайна стоимости различных типов ошибок совпадают. Однако можно также рассматривать функции расстояния с различными значениями стоимости операций редактирования. Пусть для любой пары символов α, β из $\mathcal{A} \cup \{\epsilon\}$ определена положительная стоимость $\delta(\alpha \to \beta) > 0$ замены α на β. Введение пустого символа, как уже отмечал, позволяет свести

операции удаления и добавления к операции замены. В частности, замена $\alpha \to \epsilon$ соответствует удалению α, а операция $\epsilon \to \beta$ соответствует вставке β.

Теперь рассмотрим вновь выравнивание слов. Поскольку для каждой операции замены задана стоимость, то можно вычислить стоимость выравнивания.

Определение 1.10 *Пусть $A = (u', v')$ — выравнивание строк u и v. Тогда стоимостью $\delta(A)$ выравнивания A будем называть величину:*

$$\delta(u', v') = \delta(A) = \sum_{i=1}^{l} \delta(u'_i \to v'_i)$$

Стоимость выравнивания пустых строк положим равной нулю.

С понятием стоимости тесно связано понятие оптимального выравнивания.

Определение 1.11 *Выравнивание будем называть оптимальным, если его стоимость минимальна. Стоимость оптимального выравнивания будем называть расстоянием редактирования и будем обозначать как $edist_\delta(u, v)$.*

Поскольку пустые символы не могут находиться в одинаковых позициях в строках u' и v', то максимальная их длина не превосходит $m+n$. Таким образом, количество выравниваний всегда конечно, а минимум корректно определен.

Если $\delta(\alpha \to \beta)$ тождественно равна 1, то расстояние редактирования совпадает с метрикой Левенштайна. В контексте строковых сравнений важной задачей является вычисление расстояния редактирования и построение оптимального выравнивания. Простой алгоритм решения этой задачи был независимо открыт несколькими авторами. Я приведу вариант, отличающийся своей простотой, который был предложен Вагнером и Фишером [38].

Идея, лежащая в основе алгоритма, заключается в вычислении расстояния редактирования между все более длинными префиксами слов u и v, используя значения расстояний редактирования для более коротких префиксов.

Теорема 1.1 *([38]) Пусть $E_\delta(i, j) = edist_\delta(u_1 u_2 \dots u_i, v_1 v_2 \dots v_j)$, $0 \le i \le m$, $0 \le j \le n$ — расстояние редактирования между двумя префиксами u и v длины i и j, соответственно, тогда выполняются следующие рекуррентные соотношения:*

$$E_\delta(0, 0) = 0 \tag{1}$$

$$E_\delta(i+1, 0) = E_\delta(i, 0) + \delta(u_{i+1} \to \epsilon) \tag{2}$$

$$E_\delta(0, j+1) = E_\delta(0, j) + \delta(\epsilon \to v_{j+1}) \tag{3}$$

$$E_\delta(i+1, j+1) = \min \begin{cases} E_\delta(i, j+1) + \delta(u_{i+1} \to \epsilon) \\ E_\delta(i+1, j) + \delta(\epsilon \to v_{j+1}) \\ E_\delta(i, j) + \delta(u_{i+1} \to v_{j+1}) \end{cases} \tag{4}$$

Доказательство. Докажем теорему индукцией по i и j [13]. Соотношения 2 и 3 следуют из того факта, что пустая строка представлена в выравнивании строкой из пустых символов ϵ, 1 следует из определения стоимости пустого выравнивания. Формулы 1, 2 и 3 составляют базис индукции. Докажем теперь индукционный шаг. Очевидно, что $E_\delta(i+1, j+1)$ не может быть меньше расстояния редактирования, потому что согласно формуле 4, эта величина является минимальным значением стоимости одного из трех выравниваний и не может быть меньше значения оптимального выравнивания. Докажем теперь, что она также не больше стоимости оптимального выравнивания.

По предположению индукции для всех $i' \leq i+1$ и $j' \leq j+1$ таких, что либо $i' < i+1$, либо $j' < j+1$, $E_\delta(i', j') = edist_\delta(u_1 u_2 \dots u'_i, v_1 v_2 \dots v'_j)$. Рассмотрим оптимальное выравнивание $A = (u', v')$ префиксов u и v длины $i+1$ и $j+1$, соответственно. Пусть длина выравнивания $A = (u', v')$ равна k. В силу того, что пустой символ не может занимать в обеих строках позицию k, то возможно три варианта:

- $u'_k = \epsilon, \quad v'_k = v_{j+1}$
- $u'_k = u_{i+1}, \quad v'_k = v_{j+1}$
- $u'_k = u_{i+1}, \quad v'_k = \epsilon$

Я рассмотрю только первый случай, а остальные случаи рассматриваются по аналогии.

Итак: $u'_k = \epsilon, \quad v'_k = v_{j+1}$, а

$$edist_\delta(u_1 u_2 \dots u_{i+1}, v_1 v_2 \dots v_{j+1}) \leq \min \left\{ \begin{array}{l} E_\delta(i, j+1) + \delta(u_{i+1} \to \epsilon) \\ E_\delta(i+1, j) + \delta(\epsilon \to v_{j+1}) \\ E_\delta(i, j) + \delta(u_{i+1} \to v_{j+1}) \end{array} \right\}$$

Очевидно, что $A^- = (u'_1 u'_2 \dots u'_{k-1}, v'_1 v'_2 \dots v'_{k-1})$ является выравниванием для префиксов u и v длины $i+1$ и j (Символ v_{j+1} остутствует в выравнивании A^-, потому что это крайний правый символ строки $v'_1 v'_2 \dots v'_k$. С другой стороны, крайний правый символ префикса u' равен ϵ, поэтому в строке $u'_1 u'_2 \dots u'_{k-1}$ столько же непустых символов u, что и в $u'_1 u'_2 \dots u'_k$), поэтому его стоимость не меньше $E_\delta(i+1, j)$ по предположению индукции:

$$E_\delta(i+1, j) \leq \delta(A^-)$$

С другой стороны, $\delta(A^-) = \delta(A) - \delta(\epsilon \to v_{j+1}) = edist_\delta(u_1 u_2 \dots u_{i+1}, v_1 v_2 \dots v_{j+1}) - \delta(\epsilon \to v_{j+1})$, откуда получаем следующее неравенство:

$$edist_\delta(u_1 u_2 \dots u_{i+1}, v_1 v_2 \dots v_{j+1}) \geq E_\delta(i+1, j) + \delta(\epsilon \to v_{j+1}) \geq E_\delta(i+1, j+1).$$

Таким образом, значение $E_\delta(i+1, j+1)$ не превосходит расстояние редактирования, а согласно замечанию в начале доказательства теоремы, оно также не

[13] Индукция может быть осуществлена для любого счетного направленного множества индексов с ассиметричным, транзитивным отношением $>(<)$. В докзательстве теоремы: $(i, j) < (i', j') \iff (i \leq i') \wedge (j \leq j') \wedge ((i < i') \vee (j < j')$

Таблица 7: Таблица вычисления расстояния редактирования слов **порт** и **пол**

0	1	2	3	4
1	**0**	1	2	3
2	1	**0**	**1**, (р $\to \epsilon$)	3
3	2	1	**1**, (р $\to$ л)	**2**, (т $\to$ л), (т $\to \epsilon$)

может быть меньше стоимости оптимального выравнивания. Это значит, что $E_\delta(i+1, j+1) = edist_\delta(u_1u_2 \ldots u_{i+1}, v_1v_2 \ldots v_{j+1})$, и индукционный шаг доказан.

Теорема доказана.

Алгоритм Вагнера-Фишера имеет сложность порядка $\mathcal{O}(m \cdot n)$ и требует также $\mathcal{O}(m \cdot n)$ памяти для своей работы. В процессе вычислений значения $E_\delta(i, j)$ хранятся в таблице размером $(m+1) \cdot (n+1)$. Как несложно заметить, для вычисления $i+1$-ой строки достаточно "помнить" только i-ую строку таблицы. Это упрощение играет очень важную роль, потому что в процессе сравнения двух строк именно память является решающим фактором, потому что как только таблица перестает помещаться в оперативной памяти, процесс обработки замедляется в сотни раз.

Алгоритм Вагнера-Фишера позволяет не только вычислять расстояние редактирования, но и также строить оптимальные выравнивания. Для этого необходимо отслеживать, какое значение под знаком минимума в формуле 4 реализует минимум.

Пример Рассмотрим строки **порт** и **пол** и построим для них оптимальное выравнивание на основе метрики Левенштайна. Результаты вычислений приведены в таблице 7. В последних двух строках приводятся варианты замен, реализующих минимум. Им соответствуют два варианта оптимального выравнивания стоимости два:

п **о** **р** **т**
п **о** **л** ϵ

и

п **о** **р** **т**
п **о** ϵ **л**

На практике часто функцию расстояния используют в контексте единичной функции стоимости, то есть метрики Левенштайна, Так называемая проблема коррекции строки u в строку v состоит в нахождении выравнивания с наименьшим количеством операций редактирования.

Алгоритмы нахождения оптимального выравнивания используются в программах сравнения файлов таких, как fc и diff, и поиска на неточное равенство.

В случае произвольной стоимостной функции δ, видимо, не существует алгоритма, имеющего сложность меньшую квадратичной. Для случая $\delta \equiv 1$ Мазек и Петерсон [22] придумали алгоритм, имеющий сложность $\mathcal{O}(m \cdot n / \log m)$. Согласно Майерсу [24], это единственный алгоритм, который в худшем случае

отрабатывает за время меньшее чем $\mathcal{O}(mn)$. К сожалению, издержки данного алгоритма таковы, что он дает выигрыш только на очень длинных строках [30].

Юкконен и Майерс независимо придумали алгоритм, отрабатывающий в худшем случае за время $\mathcal{O}(n \cdot edist_\delta(u,v))$ и за $\mathcal{O}(n + edist_\delta(u,v))$ — в среднем.

В качестве модели сравнения строк чаще всего используется модель расстояния редактирования. Как я уже отмечал, сложность алгоритма имеет квадратичный порядок. Поэтому были попытки создания и разработки других моделей, имеющих меньшую сложность. Одним из вариантов является модель максимального соответствия, предложенная Эренфойхтом и Хауслером [14].

Идея, лежащая в основе вычисления функции расстояния, заключается в сравнении в терминах общих подслов. Строки считаются похожими, если они обладают длинными общими подстроками. Ключевым понятием является разбиение.

Определение 1.12 *([14]) Разбиением v по u называется набор $P = (w_1, \alpha_1, w_2, \alpha_2, \ldots, w_r, \alpha_r, w_{r+1})$, где w_i — подслова слова u, α_i — символы алфавита $\mathcal{A}$, и $v = w_1\alpha_1 w_2\alpha_2 \ldots w_r\alpha_r w_{r+1}$. Число r, обозначаемое также как $|P|$, называется размером разбиения.*

Определение 1.13 *Расстоянием максимального соответствия $mmdist(u,v)$ называется минимально возможный размер разбиения.*

Пример. $u = cbaabdcb$, $u = abcba$. $P_1 = (cba, a, b, d, cb)$, $P_2 = (cb, a, ab, d, cb)$ — минимальные разбиения u по v.

Существуют два канонических разбиения.

Определение 1.14 *Пусть $P = (w_1, \alpha_1, w_2, \alpha_2, \ldots, w_r, \alpha_r, w_{r+1})$ — разбиение v по u. Если для всех i $w_i\alpha_i$ не является подсловом u, то P называется разбиением слева-направо. Если же для всех i $\alpha_i w_{i+1}$ не является подсловом u, то P называется разбиением справа-налево.*

Введенные разбиения обладают свойством минимальности и могут быть эффективно вычислены, Для этого все суффиксы u заносятся в trie-дерево (см. стр. 22), которое позволяет для любой строки w найти максимальный общий префикс u и w за $\mathcal{O}(|\mathcal{A}| \cdot |w|)$ операций, где w — длина слова, а $|\mathcal{A}|$ — размер алфавита. Для построения разбиения слева-направо, сначала определяется максимальный общий префикс $v_1 v_2 \ldots v_i$ слов u и v. Затем из слова v удаляется общий префикс и следующий за ним символ, и операция повторяется с новым словом до тех пор, пока из v не удалили все символы. Согласно замечанию о времени поиска общего префикса, построение разбиения потребует $\mathcal{O}(|\mathcal{A}| \cdot |v|)$ плюс $\mathcal{O}(|\mathcal{A}| \cdot |u|)$ операций, которые затрачиваются на построение дерева. Получается, что вычисление $mmdist(u,v)$ требует $\mathcal{O}(|\mathcal{A}| \cdot (m+n))$ операций в худшем случае, и если размер алфавита невелик, то расстояние максимального соответствие подсчитывается быстрее, чем расстояние редактирования. Оба эти расстояния связаны неравенством ([14]):

$$mmdist(u,v) \leq edist_1(u,v),$$

Наряду с моделью максимального соответствия используется модель n-грамм. В этой модели также учитываются общие подслова, однако в отличие от модели наилучшего соответствия рассматриваются только общие подслова, имеющие длину n.

Определение 1.15 *([32]) Профилем n-грамм с параметром q будем называть функцию $G_q(u)(w)$, $w \in \mathcal{A}^q$, которая равна количеству вхождений n-граммы w, $|w| = q$ длины q в слово u.*

Определение 1.16 *Расстоянием n-грамм с параметром q между u и v будем называть следующую величину:*

$$ngdist_q(u, v) = \sum_{w \in \mathcal{A}^q} |G_q(u)(w) - G_q(v)(w)|$$

Данная функция расстояния обладает свойством симметричности и удовлетворяет неравенству треугольника, однако она не является метрикой, потому что может равняться нулю на паре различных слов. Например, $ngdist_2(aaba, abaa) = 0$.

Сложность вычисления $ngdist_q(u, v)$ примерно совпадает со сложностью вычисления $mmdist(u, v)$. Как и $mmdist$ ее можно использовать для оценки расстояния Левенштайна, потому что $ngdist$ удовлетворяет следующему неравенству ([32]):

$$\frac{ngdist_q(u, v)}{2q} \leq edist_1(u, v)$$

На практике задача так называемого поиска строки, допускающего k ошибок, актуальнее, чем, собственно, задача нахождения расстояния редактирования. Одним важным частным случаем является поиск по шаблону **%строка%**. Другими словами, при заданных максимально допустимом количестве ошибок k, строкам t и p, нужно найти все вхождения подстроки v строки t такие, что $edist_\delta(p, v) \leq k$.

Я начну рассмотрение решений этой задачи с алгоритма Селлерса ([28]). Этот алгоритм является модификацией алгоритма Вагнера-Фишера, описанного на стр. 28. Основная идея Селлерса состоит в том, чтобы положить стоимость вставки символов t_i в пустую строку равной нулю. Технически это означает, что первая строка в таблице значений $E_\delta(i, j)$ равна нулю. При этом $E_\delta(i, j)$ равна минимальному расстоянию редактирования между префиксом шаблона $p_1 p_2 \dots p_i$ длины i и некоторым суффиксом $t_l t_{l+1} \dots t_j$ префикса t длины j (минимум берется по всем суффиксам строки $t_1 t_2 \dots t_j$, $E_\delta(i, j) = \min edist_\delta(s, p)$, где s — суффикс $t_1 t_2 \dots t_j$). Рекуррентные соотношения для $E_\delta(i, j)$ записываются следующим образом:

$$E_\delta(0, 0) = 0 \tag{5}$$

$$E_\delta(i + 1, 0) = E_\delta(i, 0) + \delta(p_{i+1} \to \epsilon) \tag{6}$$

$$E_\delta(0, j + 1) = 0 \tag{7}$$

$$E_\delta(i+1,j+1) = \min \begin{cases} E_\delta(i,j+1) + \delta(p_{i+1} \to \epsilon) \\ E_\delta(i+1,j) + \delta(\epsilon \to t_{j+1}) \\ E_\delta(i,j) + \delta(p_{i+1} \to t_{j+1}) \end{cases} \quad (8)$$

Выполнение условия $E_\delta(m,j) \leq k$ соответствует неточному равенству шаблона p и суффикса $t_l t_{l+1} \ldots t_j$. Как несложно видеть, изменилось только соотношение 7.

Как и в случае алгоритма Вагнера-Фишера, нет необходимости в хранении таблицы $E_\delta(i,j)$ целиком (если не нужно строить оптимальное выравнивание). Ее можно вычислять, например, столбец за столбцом. Формулы 5-8 — декларативные и не описывают порядок вычисления $E_\delta(i,j)$. Для того, чтобы можно было применить формулу перехода 8, все значения E_δ, фигурирующие в правой части, должны быть вычислены. $E_\delta(i+1,j+1)$ зависит от $E_\delta(i,j)$, $E_\delta(i+1,j)$ и $E_\delta(i,j+1)$, поэтому если рассчитывать элементы матрицы E_δ по столбцам в порядке увеличения номеров строк j, то все значения в правой части 8 будут всегда определены. Количество операций, необходимых для нахождения неполного соответствия, имеет порядок $\mathcal{O}(m \cdot n)$. Это, надо отметить, довольно высокая сложность. Чтобы уменьшить количество выполняемых операций, Юкконен ([31],[33]) придумал модификацию алгоритма Селлерса, в которой вычисляется только часть значений матрицы E_δ. Юкконен заметил, что от элементов в той части столбца j, где нет ни одного элемента, чья величина меньше либо равна k, не зависят существенные, т.е. не превышающие k, элементы в столбцах с номерами $j+1, j+2, \ldots n$.

Определение 1.17 *Пусть максимально возможное количество ошибок, которое в дальнейшем будет считаться постоянным, равно k. Пусть также v и v' — векторы-столбцы размерности $m+1$. Будем называть v и v' эквивалентными ($v \equiv v'$), если $v(i) = v'(i)$ в случае, когда $v(i) \leq k$ или $v'(i) \leq k$.*

Столбцы матрицы E будем также называть столбцами расстояний. Формулы (5)-(6) задают вид самого левого столбца расстояний, а формулы (7)-(8) — функцию перехода $v_b = nextdcol(v,b)$:

$$v_b(0) = 0 \quad (9)$$

$$v_b(i+1) = \min \begin{cases} v_b(i) + \delta(p_{i+1} \to \epsilon) \\ v(i+1) + \delta(\epsilon \to b) \\ v(i) + \delta(p_{i+1} \to b) \end{cases} \quad (10)$$

Таким образом, $E_\delta(,j+1) = nextdcol(E_\delta(,j), t_{j+1})$.

Элементы столбцов расстояний, превышающие k, играют роль только при вычислении следующего столбца матрицы, потому что неполное соответствие может быть только в случае, когда элемент матрицы как раз не превышает k. Кроме того, свойство эквивалентности сохраняется при вычислении $j+1$-го столбца через j-ый и можно заменить E_δ на некоторую эквивалентную матрицу E'_δ, и при этом множества столбцов, j, для которых m-ый элемент не больше k для E_δ и E'_δ будут совпадать. Также будут совпадать значения элементов

меньшие либо равные k. Это значит, что в контексте задачи поиска неточных соответствий строки p в строке t эти матрицы — взаимозаменяемы. Верна следующая лемма о сохранении эквивалентности операцией *nextdcol*:

Лемма 1.1 *[33] Пусть* $v \equiv v'$, *тогда* $nextdcol(v, b) \equiv nextdcol(v', b)$

Доказательство. Докажем лемму индукцией по i. Для $i = 0$ $v(i) = v'(i) = 0$ (формула 9). Это равенство составляет базис индукции — докажем теперь индукционный переход от i к $i + 1$. Предположим теперь, что $v_b = nextdcol(v, b) \leq k$, и рассмотрим следующие случаи:

- $v_b(i + 1) = v_b(i) + \delta(p_{i+1} \to \epsilon) \leq k$, тогда $v_b(i) \leq k$ и $v_b(i) = v'_b(i)$ по предположению индукции.
- $v_b(i+1) = v(i+1) + \delta(\epsilon \to b) \leq k$, тогда $v(i+1) = v'(i+1)$, в силу условий $v \equiv v'$ и $v(i + 1) \leq k$.
- $v_b(i+1) = v(i) + \delta(p_{i+1} \to b) \leq k$, тогда, аналогично предыдущему пункту, $v(i) = v'(i)$.

Теперь, воспользовавшись 10, получаем, что $v'_b(i + 1) \leq v_b(i + 1) \leq k$. Проводя аналогичные рассуждени, но отталкиваясь от факта $v'_b(i + 1) \leq k$, получаем, что $v_b(i + 1) \leq v'_b(i + 1) \leq v_b(i + 1)$, то есть

$$v_b(i + 1) = v'_b(i + 1),$$

что доказывает индукционный переход. **Лемма доказана.**

Определение 1.18 *[33] Пусть* v — *столбец, тогда* $v(i)$ *является существенным элементом, если* $v(i) \leq k$, $lei(v) = max(i,\ 0 \leq i \leq m, v(i) \leq k)$ — *максимальный существенный индекс элемента*[14].

Заметим, что $v(m) \leq k$ выполняется тогда и только тогда, когда $lei(v) = m$, т.е. m — максимальный существенный индекс. По лемме 1.1 существенные элементы $nextdcol(v, b)$ не зависят от несущественных элементов v. Действительно, предположим, что это не так, тогда существует существенный элемент с номером i и эквивалентный столбец v' такие, что $nextdcol(v, b)(i) \neq nextdcol(v', b)(i)$, а этого не может быть, так как операция *nextdcol* сохраняет эквивалентность и, следовательно, равенство существенных элементов.

Это значит, что не обязательно вычислять $E_\delta(, j+1) = nextdcol(E_\delta(, j), t_{j+1})$ целиком, как в алгоритме Селлерса. Вычисление $E_\delta(, j + 1) = nextdcol(E_\delta(, j))$ может быть модифицировано следующим образом: вычисляем $E_\delta(0, j+1), E_\delta(1, j+1), \ldots E_\delta(l, j + 1)$, где $l = lei(E_\delta(, j))$, в соответствии с формулами $9 - 10$. Далее по формулам:

$$E_\delta(l + 1, j + 1) = \min \begin{cases} E_\delta(l, j + 1) + \delta(p_{l+1} \to \epsilon) \\ E_\delta(l, j) + \delta(p_{l+1} \to t_{j+1}) \end{cases}$$

[14]LEI - The Largest Essential Index.

$$E_\delta(l+2, j+1) = E_\delta(l+1, j+1) + \delta(p_{l+2} \to \epsilon)$$

...

до тех пор пока не мы не дойдем до индекса m, либо не получим $E_\delta(h, j+1) > k$, для какого-либо индекса h. Действительно, поскольку $\delta > 0$, то значения элементов столбца расстояний $E_\delta(h, j+1)$ с номерами h большими l монотонно возрастают по h, поэтому возможно два варианта: $lei(E_\delta(, j+1)) = m$, или $lei(E_\delta(, j+1)) = h-1$. Таким образом, как только найден максимальный существенный индекс $lei(j+1)$, можно завершать вычисление столбца $j+1$. Неполное соответствие, как я уже отмечал, соответствует случаю равенства максимального существенного индекса m.

Этот алгоритм был предложен Юкконеном ([31]) для случая $\delta \equiv 1$. В среднем, количество операций для данного алгоритма имеет порядок $\mathcal{O}(k \cdot n)$, как для произвольной положительной, так и единичной функции стоимости δ.

На практике все описанные алгоритмы, основанные на вычислении матрицы E_δ, работают медленно. Это особенно актуально в случае, когда число ошибок k мало. К счастью, существуют другие методы, которые дают выигрыш в скорости на один-два порядка при поиске на основе единичной функции стоимости δ. Эти методы, реализованные в программе agrep ([36]), работают гораздо быстрее большинства методов, в том числе, основанных на алгоритмах Юкконена [30],[31].

Создатели agrep адаптировали метод Баэца-Йатеса и Гоннета ([9]) для поиска на неточное соответствие. Практическая ценность этого алгоритма заключается в том, что он использует битовые массивы, фактически распараллеливая вычисления на одном процессоре. В большинстве современных компьютеров длина машинного слова не меньше 32 битов. Это означает, что за одну операцию побитового $\wedge$ процессор производит, скажем, 32 операций $\wedge$. Операция побитового сдвига $Rshift(v)$, определяемая как

$$u = Rshift(v), \text{ где}$$

$$u(0) = 1, \quad u(i) = v(i-1),\ i > 0,$$

и осуществляющая целую серию присваиваний, выполняется также быстро, как сложение целых чисел.

Рассмотрим для начала алгоритм Баэца-Йатеса и Гоннета, осуществляющий "точный" поиск подстроки p в тексте t. Пусть $R_j(i)$ — битовая матрица размером $(m+1) \times (n+1)$, где m равно длине строки p, а n — длине текста t. Столбец R_j содержит информацию о вхождении всех префиксов p в строку t, заканчивающихся в позиции j, а именно, $R_j(i) = 1$, если $p_1 p_2 \dots p_i = t_{j-i+1} t_{j-i+2} \dots t_j$, то есть когда i символов p совпадают с i символами t до j-го включительно. Считывая t_{j+1}, необходимо проверять не появились ли новые вхождения префиксов большей длины. Для каждого i такого, что $R_j(i) = 1$, нужно проверять совпадают ли t_{j+1} и p_{i+1}. Если $R_j(i) = 0$, то префикс $p_1 p_2 \dots p_i$ не совпадает с $t_{j-i+1} t_{j-i+2} \dots t_j$ и $p_1 p_2 \dots p_{i+1}$ не может совпадать с $t_{j-i+1} t_{j-i+2} \dots t_{j+1}$. Если $t_{j+1} = p_1$, то $R_{j+1}(1) = 1$. Если $R_{j+1}(m) = 1$, то это означает, что p входит в t, занимая позиции с $j-m+2$ по $j+1$.

Таблица 8: Поисковая матрица R для строк **вал** и **завал**.

	з	а	в	а	л
в	0	0	1	0	0
а	0	0	0	1	0
л	0	0	0	0	1

Зависимость столбца R_{j+1} от R_j выражается следующими соотношениями:

$$R_j(0) = 1, \quad \forall j \tag{11}$$

$$R_0(i) = 0, 1 \leq i \leq m \tag{12}$$

$$R_{j+1}(i+1) = \begin{cases} 1, R_j(i) = 1 \text{ и } p_{i+1} = t_{j+1} \\ 0, \text{в противном случае} \end{cases} \tag{13}$$

Это преобразование, если производить его отдельно для каждого элемента матрицы, требует такого же количества операций, что и алгоритм Селлерса. Однако, поскольку мы здесь имеем дело с битовыми массивами, весь столбец R_j можно вычислить с помощью нескольких операций следующим образом. Рассмотрим алфавит $\mathcal{A} = \{\alpha_1, \alpha_2, \ldots \alpha_\Sigma\}$. Для каждого символа α_i вычислим битовый массив S_{α_i} размером m такой, что у него единице равны биты в позициях r, если r-ый символ строки p равен α_i Другими словами, S_{α_i} содержит маску вхождений символа α_i в строку p. Достаточно построить такие массивы только для символов, входящих в p, потому что для всех остальных они просто равны нулю.

Поскольку нулевой элемент каждого столбца матрицы R всегда равен единице мы не будем хранить его в R, уменьшив размер столбца R на один бит. Как несложно проверить, переход от R_j к R_{j+1} осуществляется с помощью одной операции сдвига вправо и побитового $\wedge$ с $S_{t_{j+1}}$. Поскольку нулевой элемент столбцов R, который мы решили не хранить в R_j, всегда равен 1, правый сдвиг должен заполнять первый бит единицей (поскольку на его месте при сдвиге должен оказываться $R_j(0)$), что полностью соответствует определению $Rshift$:

$$R_{j+1} = Rshift(R_j) \wedge S_{t_{j+1}} \tag{14}$$

Пример. Пусть нужно найти подстроку **вал** в строке **завал**. Битовые массивы S_i для букв **в, а, л** равны $100, 010, 001$. Матрица преобразований R приведена в таблице 8. $R_5(3) = 1$, что соответствует вхождению строки **вал** в **завал**.

Разумеется, алгоритм можно реализовать только в том случае, когда m не превышает размер машинного слова. Если, например, m равно размеру двух машинных слов, то необходимо представлять каждый R_j и S_{t_i} двумя

машинными словами, а для перехода от R_j к R_{j+1} потребуется примерно четыре операции: два сдвига с переносом значения последнего бита одного машинного слова в первый бит другого и две операции побитового $\wedge$.

Заметим, однако, что почти все время, вторая половина битового массива R_j будет равна нулю, потому что вхождения образца длиной в машинное слово в случайный текст крайне редки. Вероятность такого вхождения меньше одной тысячной для алфавита размером в 20-30 символов, поэтому почти все время достаточно производить вычисления только с частью R_j, помещающейся в одно слово, и использовать дополнительные машинные слова, только когда правый крайний бит становится равным единице. Кроме того, поисковые образцы длиннее, например, 32 символов — нечастое явление.

Алгоритм Баэца-Йатеса и Гоннета легко реализуем и в предположении, что длина p не превышает размер машинного слова, в худшем случае требует $\mathcal{O}(n)$ операций. На практике он работает почти так же быстро, как алгоритмы Кнута-Морриса-Прата и Бойера-Мура [15] ([13]).

Рассмотрим теперь, как использовать этот алгоритм для поиска подстроки в t, отличающейся от p не более чем на k ошибок. Хочу сразу отметить, что как и метод Селлерса, алгоритм agrep'а использует метод динамического программирования. Однако вместо того, чтобы вычислять матрицу $E_1(i,j)$, содержащую значения расстояний редактирования между $p_1p_2\dots p_i$ и некоторым суффиксом $t_1t_2\dots t_j$, в алгоритме используется $k+1$ матриц R^d, $0 \le d \le k$ размером $(m+1)\cdot(n+1)$ таких, что $R^d_j(i) = 1$, если $\min_l(edist(p_1p_2\dots p_i, t_lt_{l+1}\dots t_j)) \le d$. Это значит, что j-ая строка матрицы R^d содержит информацию о наличии соответствий префиксов p суффиксам строки $t_1t_2\dots t_j$ с не более чем d ошибками. Хотя вычисление $j+1$-го столбца R^d через j-ые и $j+1$-ые столбцы матриц $R^d, R^{d-1}, \dots R^0$ полностью аналогично переходу от j-ого к $j+1$-ому столбцу матрицы E_δ, $(\delta \equiv 1)$, за счет использования быстро вычисляемых операций побитового $\wedge$ и правого сдвига достигается существенный выигрыш в скорости.

Для выведения рекуррентной формулы, выпишем формулы Селлерса 5-8 для случая $\delta \equiv 1$.

$$E_1(i,0) = i$$

$$E_1(0,j) = 0$$

$$E_1(i+1,j+1) = \min \begin{cases} E_1(i,j)+1, \ t_{j+1} \neq p_{i+1}, \text{ замена} \\ E_1(i,j), \ t_{j+1} = p_{i+1} \\ E_1(i,j+1)+1, \text{ удаление } p_{i+1} \\ E_1(i+1,j)+1, \text{ вставка } t_{j+1} \end{cases}$$

Из последней формулы следует, что $E_1(i+1,j+1)$ может равняться d только в следующих случаях:

- $E_1(i,j) = d-1$, этот случай соответствует замене p_{i+1} на t_{j+1}.

[15] Разница в скорости составляет 10-20%% и, видимо, может слегка варьироваться в зависимости от реализации в ту или другую сторону.

- $E_1(i,j) = d$, т.е. префикс p длины i соответствует суффиксу $t_1 t_2 \dots t_{j+1}$ с не более чем d ошибками, а символы t_{j+1} и p_{i+1} совпадают.
- $E_1(i,j+1) = d-1$, — удалению p_{i+1}.
- $E_1(i+1,j) = d-1$, — вставка t_{j+1}.

Таким образом,

$$E_1(i+1,j+1) = d \Longleftrightarrow$$
$$((E_1(i,j) = d) \wedge (t_{j+1} = p_{j+1})) \vee$$
$$(E_1(i,j+1) = d-1) \vee (E_1(i,j) = d-1)) \vee (E_1(i+1,j) = d-1) \qquad (15)$$

Формула 15, в отличие от формул Селлерса, позволяет использовать для вычислений матрицы R^d. Если мы, как в случае алгоритма точного равенства, не будем хранить нулевую строку R^d, то формулы преобразования записываются следующим образом:

$$R_0^d = 11 \dots 0,\ d \text{ единиц}$$
$$R_{j+1}^d = (Rshift(R_j^d) \wedge S_{t_{j+1}}) \vee Rshift(R_j^{d-1}) \vee Rshift(R_{j+1}^{d-1}) \vee R_j^{d-1},$$

где битовый массив $S_{t_{j+1}}$ содержит единицы в позиции r только если $p_r = t_{j+1}$.

$$R_{j+1}^d = (Rshift(R_j^d) \wedge S_{t_{j+1}}) \vee Rshift\left(R_j^{d-1} \vee R_{j+1}^{d-1}\right) \vee R_j^{d-1} \qquad (16)$$

Получается, что для вычисления каждого R^d необходимо выполнить два сдвига, три побитовых $\vee$ и одно побитовое $\wedge$, а общая сложность алгоритма составляет $\mathcal{O}((k+1)\cdot(n+m))$. Казалось бы, среднее время алгоритмов Юкконена имеет тот же порядок, однако, как показывают результаты тестов ([35]), agrep работает в несколько раз быстрее.

Одной из причин такого высокого быстродействия является использование "фильтра" для поиска позиций в тексте, где возможно неточное соответствие. Работа этого фильтра основана на следующем свойстве. Пусть $p = P_1 P_2 \dots P_{k+1} P'$ представлен в виде последовательно идущих подстрок длины $\left\lfloor \frac{m}{k+1} \right\rfloor$[16] и, возможно пустого, суффикса P'. Тогда если $edist_1(p,w) \leq k$, то одна из подстрок P_i является также подстрокой w. Действительно, если ни одно из P_i не является подстрокой w, то в оптимальном выравнивании (p', w') в каждой подстроке P_i будет символ $p'_k \in p'$, такой что $p'_k \neq w'_k$, например в случае, когда $w'_k = \epsilon$ (если это не так, то P_i будет, очевидно, подстрокой w). Но это означает, что стоимость выравнивания будет не меньше числа подстрок P_i, а именно, $k+1$, что противоречит предположению $edist_1(p,w) \leq k$.

Таким образом, нужно искать вхождения строк P_i в t и, когда найдено хотя бы одно из них, дополнительно искать, используя вышеописанный стандартный алгоритм в окрестности размером m символов.

Этот вариант работает быстро только в случае небольшого по сравнению с m числа k. Чтобы "фильтр" работал эффективно, длина строк P_i должна быть

[16] $\lfloor x \rfloor$ — наименьшее целое число, не превышающее x.

не меньше трех, т.к. частота появления три-граммы варьируется от 0.0005 до 0.005, а для би-граммы она уже может составлять 0.01-0.05. Если $m = 2(k+1)$, то нужно дополнительно проверить окрестность примерно каждого тридцатого символа, а при длине p порядка пяти-десяти символов выигрыш получается совсем небольшим.

Основное преимущество данной модификации состоит в том, что алгоритм Баэца-Йатеса и Гоннета можно модифицировать так, чтобы искать по всем P_i одновременно! Я проиллюстрирую это на примере $p = ABCDEFGHIJKL$, $m = 12$, $k = 3$, $P_1 = ABC$, $P_2 = DEF$, $P_3 = GHI$, $P_4 = JKL$. Необходимо проверять вхождения каждого P_i в текст. Сформируем комбинированный поисковый шаблон путем перемешивания букв, выбирая сначала первую букву P_1, P_2, P_3, P_4, затем снова первую букву P_1, и.т.д., пока не выберем все буквы. При этом получится шаблон: $ADGJBEHKCFIL$. Далее вычисляется для этого нового шаблона набор битовых масок S_{α_i} в точности как в стандартном алгоритме Баэца-Йатеса и Гоннета. Матрица R считается аналогичным образом с одним лишь отличием: вместо сдвига на один бит, происходит сдвиг на $k+1 = 4$ бита с заполнением первых $k+1$-х битов единицами:

$$R_{j+1} = (Rshift)^{k+1} R_j \wedge S_{t_{j+1}}$$

При этом, если один из $k+1$-х последних битов равен 1, то одно из P_i входит в t. Это условие проверяется с помощью одной операции побитового $\wedge$ и операции сравнения с нулем. Поскольку $k+1$ одна операция правого сдвига и заполнение первых $k+1$ битов единицами занимает примерно столько же времени, что и одна операция $Rshift$, то сложность алгоритма фильтрации равна $\mathcal{O}(n)$.

Сильной стороной алгоритма agrep является его гибкость и расширяемость. Его можно адаптировать для нахождения соответствий по всей строке[17], поиска по несложным регулярным выражениям, в которых используются спецсимволы, обозначающие произвольную подстроку (%), произвольный символ (_) и набор символов ([Abz]). Более того, он может быть адаптирован, хотя и в довольно ограниченной степени, для поиска с неединичной функцией стоимости δ, а именно, когда стоимости замены, удаления и вставки являются небольшими целыми числами.

Создатели agrep не рассматривают в своей статье [36] случая соответствия всей строке t, поэтому я сначала рассмотрел аналог алгоритма Селлерса, однако этот важная задача также решается с помощью преобразований битовых матриц.

Пусть теперь, как и в случае алгоритма Вагнера-Фишера, $E_1(i,j) = edist_1(p_1p_2 \ldots p_i, t_1t_2 \ldots t_j)$ — расстояние редактирования между префиксами p и t длины i и j соответственно, а $R_j^d(i) = 1 \iff E_1(i,j) \leq k$. В данном случае столбец R_j^d описывает уже соответствие между префиксами p и всей строкой $t_1t_2 \ldots t_j$ с точностью до d ошибок. Однако, что самое важное, формула вычисления R_{j+1}^d, через R_j^d для полной, содержащей нулевую строку

[17] т.е. для поиска строк t, таких что $edist_1(p,t) \leq k$, для всей строки t, а не только для некоторого подслова w.

матрице, в точности совпадают с формулой 16. Это следует из идентичности рекуррентных формул для $E_\delta(i,j)$

Еще одним интересным аспектом применения алгоритма agrep является поиск по простым регулярным выражениям. Рассмотрим, например, шаблон $p = p_1$[a-e]p_2. Пусть выражение для набора символов [a-e] занимает i-ую позицию в шаблоне p, тогда единственное изменение, которое необходимо произвести: в каждом битовом массиве $S_a, S_b, \ldots S_e$ присвоить биту номер i значение 1. Это будет означать, что алгоритм допускает в позиции i любой из символов a, b, c, d, e. Аналогичным образом можно поступить и с символом _ , обозначающим произвольный символ.

Алгоритм может быть также модифицирован, чтобы обрабатывать шаблоны, содержащие символ %, но это несколько сложнее. Случай % подробно описан в статье [36], доступной на сайте ftp://ftp.cs.arizona.edu.

1.4.4 Индексирование словаря

1. Метод n-грамм и алгоритм Дж. Вилбура Идея этого метода заключается в следующем: если строка v "похожа" на строку u, то у них должны быть какие-либо общие подстроки. Более того, если расстояние Левенштайна между u и v не превышает k : $L(u,v) \leq k$, то если представить строку u в виде последовательности из $k+1$-ой строки: $u = u_1 u_2 \ldots u_{k+1}$, то хотя бы одна из u_i является подстрокой v.

Это полезное свойство наводит на мысль об организации словаря в виде инвертированного файла, в котором роль терминов играют подстроки фиксированной длины, например два и/или три. Пусть, например, в качестве терминов используются три-граммы, которые будем также называть триадами, тогда, инвертировав словарь, можно осуществлять поиск следующим образом: для ключевого слова запроса строить множество всех триад t_i этого слова и для каждой t_i последовательно считывать все ключевые слова, содержащие данную триаду, возвращая пользователю только те термины, которые "похожи" на термины запроса.

К сожалению, в такой организации данных есть свои минусы: для каждого указателя в инвертированном списке n-грамм нужно считывать ключевое слово из словаря. Поскольку таких ключевых слов относительно немного: их доля составляет порядка одной пятисотой от общего количества терминов, а "разбросаны" они по словарю случайно, то для прочтения каждого термина, почти всегда нужно считывать целый блок размером 1-4K! Таким образом, чтение только одного списка приводит к считыванию значительных объемов данных (порядка одной десятой от исходного размера словаря). Кроме того, считывание осуществляется с большим количеством позиционирований, а скорость произвольного доступа в несколько раз меньше чем последовательного. То есть выигрыш от сокращения количества дисковых операций практически полностью "съедается" потерей в скорости считывания и эксперимент доказывает это.

Эту проблему, насколько мне известно, можно решить двумя способами: хранить вместо указателя ключевое слово целиком, либо хранить в индексе некоторые статистические данные, позволяющие эмпирическим путем отсечь "бесперспективные" термины без обращения к словарю. Первый метод, видимо, является самым быстрым из существующих на сегодня. Данные читаются, практически, без позиционирований и их объем составляет несколько процентов от исходного. Возможно, что он же и самый избыточный по объему индекса. Без применения методов сжатия, индекс, как минимум, в пять-шесть раз превышает размер словаря. Если сами ключевые слова еще и могут быть сжаты в два-три раза, то для указателей на списки вхождений ключевых слов в документы степень сжатия будет явно меньше. Еще одна проблема связана с поиском по коротким терминам. Пусть пользователь ввел поисковый термин: **пагом**, ошибочно написав в слове паром г вместо р. Нетрудно видеть, что у слов нет ни одной общей подстроки длины три! Получается, что иногда нужно еще хранить и двух-буквенные начала слов. Однако если пользователь переставит буквы а и р, то это слово, опять-таки, не будет найдено. Кроме того, процент вхождения подстроки из двух букв в слова намного выше, а это снижает скорость поиска.

Второй способ был придуман Джоном Вилбуром[18] (исходники и краткое описание алгоритма были любезно предоставлены Олегом Ховайко [19]). В этом алгоритме для каждой триады хранится ее общий вес Gw в словаре равный логарифму отношения числа вхождений триады в словарь к числу вхождений триады в слово, а также вес этой триады в каждом ключевом слове Lw равный логарифму числа вхождения триады в ключевое слово плюс единица:

$$Gw(t) = \log(N/N_t), \quad Lw(t) = \log(n+1)$$

Процесс поиска полностью аналогичен описанному варианту с тем отличием, что для каждого термина u, содержащего хотя бы одну триаду t_j, вычисляется функция соответствия u термину запроса v:

$$M(u,v) = \sum_{t_j} Lw_u(t_j) \times Lw_v(t_j) \times Gw(t_j),$$

где $Lw_s(t)$ — локальный вес триады t в ключевом слове s, равный логарифму отношения числа вхождений t в слово к общему числу триад в слов, а t_j — общая триада слов u и v. Если значение $M(u,v)$ превышает некоторое пороговое значение ϵ, то термин добавляется в список найденных, либо производится дополнительная проверка (например, что $L(u,v) \leq 2$). Несложно видеть, что в данном методе обращения к словарю чрезвычайно редки.

С моей точки зрения это очень неплохой метод, который выдает пользователю найденные ключевые слова отсортированными по степени соответствия. Кроме того, небольшая модификация этого алгоритма позволяет существенно ускорить поиск по часто используемым терминам. Тем не менее, здесь также нельзя обойтись без хранения подстрок меньшей длины. В рассматриваемой мной

[18]Вероятно существует множество аналогичных способов.

[19]http://olegh.spedia.net

модификации этого алгоритма дополнительно вводятся две синтетические триады: префиксы термина длины один и два, соответственно. Обе этих триады в силу их небольшого размера обладают довольно большими инвертированными списками, что снижает скорость поиска.

2. Древовидные структуры, ускоряющие поиск. Как уже было сказано ранее, поиск на неточное равенство в текстовом информационном массиве может быть сведен к поиску в словаре ключевых слов этого массива, потому что, как правило, размер словаря в сотни раз меньше суммарного размера документов. Если документы состоят по большей части из текстов на естественных языках или языках программирования, то размер словаря ограничен количеством слов языка. Тем не менее, бывают ситуации, когда размер этого словаря составляет миллионы и даже десятки миллионов записей. Например, если поисковая система содержит информацию о местонахождении в сети Интернет различных файлов на FTP серверах, то это число может быть очень большим. В настоящее время существуют системы[20], содержащие сведения примерно о сотне миллионов файлов. Разумеется, что один и тот же архив хранится на нескольких серверах, но даже с учетом этого условия, словарь может содержать порядка 2-10 миллионов записей.

Очевидно, что какие-то методы сокращения перебора просто необходимы. В настоящее время существует множество подобных методов, которые можно разделить на методы условно называемые мною методы n-грамм (они описаны в предыдущем подразделе) и методы, использующие неравенство треугольника. Рассмотрим пример пространства $\mathcal{U}$ с метрикой $\varrho(x, y)$ и единичным шаром с центром в точке ноль, содержащим конечное множество точек $\mathcal{U}$. Пусть теперь необходимо определить может ли точка y быть одной из точек множества $\mathcal{U}$, попадающих в единичный шар, при условии, что точность задания ее положения равна ϵ в смысле метрики ϱ. Множество $\mathcal{U}$ конечно, поэтому задачу можно решить перебором, однако можно и ускорить процесс поиска.

Выберем покрытие шара некоторым конечным набором сфер (x_i, r_i), $1 \leq i \leq n$ (x_i — центр, r_i — радиус), и вычислим n расстояний от y до каждой из точек x_i. Пусть теперь для какого-либо i:

$$\varrho(x_i, y) > \epsilon + r_i,$$

тогда в силу неравенства треугольника внутри сферы с центром x_i радиуса r_i не может быть искомых точек.

Эту идею можно реализовать на практике множеством способов, я же опишу алгоритм построения trie-деревьев Андрю Бермана ([11]).

Будем для простоты считать, что функция ϱ принимает только целочисленные значения, что в большинстве случаев совершенно естественно. Выберем n "ключевых" представителей исходного множества $k_1, k_2, \ldots, k_n \in \mathcal{U}$. Для каждого элемента $x \in \mathcal{U}$ посчитаем n расстояний $\varrho(x, k_i)$ и построим из них $n-$компонентный вектор $r(x)$, $r(x)_i = \varrho(x, k_i)$, компоненты которого равны расстояниям до ключевых элементов. Полученное множество векторов

[20]Например Lycos, см. http://ftpsearcg.lycos.com

Таблица 9:

Корень $\rightarrow$	0 $\rightarrow$	1 $\rightarrow$	подог
	1 $\rightarrow$	0 $\rightarrow$	подох
		2 $\rightarrow$	порог
	2 $\rightarrow$	1 $\rightarrow$	порох

организуем в виде trie-дерева. Trie-дерево строится совершенно аналогично строковому trie-дереву (см. стр. 22). Я проиллюстрирую эту структуру на примере. Пусть $\mathcal{U} = (\text{порог}, \text{подог}, \text{подох}, \text{порох})$ и $\varrho(x, y) = edist_1(x, y)$, то есть в качестве метрики выбрано расстояние Левенштайна. Пусть $n = 2$, $k_1 = \text{подог}$, $k_2 = \text{подох}$. Тогда:

$r(\text{порог}) = (1, 2)$

$r(\text{подог}) = (0, 1)$

$r(\text{подох}) = (1, 0)$

$r(\text{порох}) = (2, 1)$

Соответствующее trie-дерево приведено в таблице 9.

Разбиение по поддеревьям на верхнем уровне производится по значениям первого компонента вектора $r(x)$, далее по значениям второго, и.т.д. Поиск в таком дереве осуществляется следующим образом: для элемента поискового шаблона p строится вектор $r(v)$, далее производится обход дерева, начиная с вершины. Если для некоторой вершины B уровня i выполняется соотношение $s - \epsilon > r(v)_i$, где s — расстояние от поискового термина до ключевого элемента k_i, то в силу неравенства треугольника все элементы поддерева с корнем в B находятся от поискового термина на расстояние большем ϵ, то есть в этом поддереве нет искомых вершин и его можно не просматривать.

Используя дерево из таблицы 9, рассмотрим пример поиска слова **порог** с точностью до одной ошибки ($\epsilon = 1$). Вектор расстояний у слова **порог**, содержащий расстояния до ключевых элементов равен $r(\text{порог}) = (1, 2)$.

Это означает, что ветку дерева, содержащую элементы, у которых первый компонент вектора $r(x)$ равен нулю, можно исключить из рассмотрения.

В приведенном примере, выигрыш невелик, но если количество элементов $\mathcal{U}$ велико, то возможно сокращение перебора более чем на порядок. Основным недостатком данного метода является необходимость чтения страниц trie-дерева в режиме произвольного доступа. Тут, несомненно, можно использовать модификации с хранением верхушки дерева в памяти (аналогично B^+-деревьям), и они, видимо, могут быть успешно реализованы, однако мне неизвестно какой выигрыш по времени можно получить за счет такой организации данных. Trie-дерево имеет смысл использовать в случае, когда операция сравнения ресурсоемка, а время ее выполнения может во много раз превышать время ввода-вывода.

Литература

[1] Разработка новых технологий информационного поиска. И. Гринберг, Л. Гарбер. Открытые системы N 10 1999, стр 28-30. (Searching for new search technologies, Ilan Greenberg and Lee Garber, — IEEE Computer, August 1999, pp. 6-11.)

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

[3] Д. Кнут, Искусство программирования, Т3.

[4] Э. Озкархан, Машины баз данных и управления базами данных, М. Мир 1989.

[5] В. Феллер, Введение в теорию вероятностей, М. 1967.

[6] Stefan Kurtz, Fundamental Alogorithms for declarative pattern matching. University Bielefeld, 1995, Ph.D Thesis (Report 95-03). (http://algo.4u.ru)

[7] Latent Semantic Indexing is an Optimal Special Case of Multidimensional Scaling, Bryan T. Bartell.

[8] Protein Sequence Alignment and Database Scanning,
In: Protein Structure Prediction: A Practical Approach, (Ed. M. J. E. Sternberg), IRL Press at Oxford University Press. ISBN 0 19 963496 3. (http://barton.ebi.ac.uk/new/publications/bookchapters.html)

[9] Baeza-Gates R.A., and G.H. Gonnet, A new approach to text searching, Proceedings of the 12th Annual ACM-SIGIR conference on Information Retrieval, Cambridge, MA (June 1989),pp. 168-175.

[10] Baeza-Gates R.A, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed-quieris trees. In Combinatorial Pattern Matching, June 1994.

[11] A new data structure for fast approximate matching. Andrew Berman. 3/94. (http://algo.4u.ru)

[12] Damashek. Method of retrieving documents that concern the same topic. Patent No. US5418951. (IBM Patent Server: http://www.patents.ibm.com)

[13] Handbook of computer science and enineering. Chapter 6. Pattern matching and text compression algorithms.

[14] A. Ehrenfeucht, D. Haussler. A New Distance Metric on Strings Computable in Linear Time. Discrete Applied Mathematics, 20:191-203, 1988.

[15] Christos Faloutsos and Douglas Oard, Survery of Information Retrieval and Filtering Methods, University of Maryland.

[16] Paolo Ferragina. Dynamic Data Structures for String. Ph.D. Thesis: TD-3/97, (http://www.di.unipit.it/~ferragin/ferragin.html)

[17] Inverted Files Versus Signature Files for Text Indexing. Justin Zobel and Alistair Moffat and Kotagiri Ramamohanarao, Collaborative Information Technology Research Institute, Departments of Computer Science, RMIT and the University of Melbourne, Australia, feb 1995, Technical report No TR-95-5

[18] LUHN, H.P., A statistical approach to mechanised encoding and searching of library information, IBM Journal of Research and Development, 1, 309-317 (1957).

[19] String Matching in Lempel-Ziv Compressed Strings Martin Farach, Mikkel Thorup. Rutgers University University of Copenhagen.

[20] Let Sleeping Files Lie: Pattern Matching in Z-Compressed Files Amihood Amir, Gary Benson, Martin Farach. Georgia Tech U. of Southern California DIMACS October 20, 1993

[21] Manber U., A text compression scheme that allows fast searching directly in the compressed file, technical report 93-07, Department of Computer Science, University of Arizona (March 1993). (ftp://ftp.cs.arizona.edu)

[22] U. Masek, M. S. Peterson. A faster algorithm for computing string-edit distances. Journal of Computer and System Sciences, 20(1), 785-807,1980.

[23] E.W. Myers, An $\mathcal{O}(W \cdot D)$ differences algorithm. Algorithmica, 2(1), 251-266,1986.

[24] E.W. Myers. An overview of sequence comparison algorithms in molecular biology. Technical report TR 91-29, University of Arizona, Tucson, Department of Computer Science, 1991.

[25] Porter M.F., An Algorithm For Suffix Stripping Program 14 (3), July 1980, pp. 130-137.

[26] C.J. van Rjisbergen, Information Retrieval, London: Butterworths, 1979 (http://www.dcs.gla.ac.uk/ir/new/pages/IR_Publi.html)

[27] G. Salton and C. Buckley, Term-weighting approaches in automatic text retrieval, Information Processing and Management, 1988,volume 24,number 5,pages 513-523.

[28] P.H. Sellers, The Theory of Computation of Evolutionary Distances: Pattern recognition. Journal of Algorithms, 1:359-373, 1980.

[29] The TELLTALE Dynamic Hypertext Environment: Approaches to scaleability. Claudia Pearce, Ethan Miller, Intelligent hypertext, 1997, 109-130

[30] E. Ukkonen, Algorithms for approximate string matching, 1985, Information and Control, 64, 100-118.

[31] E. Ukkonen, Finding approximate patterns in strings, $\mathcal{O}(k \cdot n)$ time. Journal of Algorithms 1985, 6 , 132-137.

[32] E. Ukkonen, Approximate String Matching with q-Grams and maximal matches. Theoretical Computer Science, 92(1), 191-211,1992.

[33] E. Ukkonen, Approximate String Matching over Suffix-Trees. In [ACGM93] pp. 229-242, 1993.

[34] Hugh E. Williams and Justin Zobel, Compressing Integers for Fast File Access, Computer Journal, Volume 42, Issue 03, pp. 193-201.

[35] Wu S. and U. Manber, Agrep — A Fast Approximate Pattern-Matching Tool, Usenix Winter 1992 Technical Conference, San Francisco (January 1992), pp. 153-162. (ftp://ftp.cs.arizona.edu)

[36] Wu S., and U. Manber, Fast Text Searching With Errors, Communications of the ACM 35 (October 1992) pp. 83-91. (ftp://ftp.cs.arizona.edu)

[37] Wu S., U. Manber, GLIMPSE: A Tool to Search Through Entire File Systems, Winter USENIX Technical Conference, 1994 (ftp://ftp.cs.arizona.edu)

[38] R.A. Wagner and M.J. Fisher, The String to String Correction Problem. Journal of the ACM, 21(1):168-173, 1974.