Алгоритмические методы доступа к словарям морфологического процессора

Егошина А.А.
Донецкий государственный институт искусственного интеллекта
postmaster@iai.donetsk.ua

Abstract

A.Yegoshina. Algorithmic methods of access to the dictionaries of the morphological processor. *The most widespread methods of search are considered and analysed. According to structure of the static dictionary of the morphological processor of information and search systems with the natural - language interface the optimum methods of search in the dictionary are chosen. The generalized algorithm of search in dictionary of morphemes of word is proposed.*

Введение

Одним из направлений интеллектуализации САПР является применение поисковых систем с естественно-языковым интерфейсом. Использование компьютерной морфологии является целесообразным в интеллектуальных системах (ИПС), реализующих поиск и анализ информации на естественном языке [1]. Основной функцией подсистемы морфологического анализа является сопоставление словоформе набора значений ее морфологических характеристик (часть речи, род, число, падеж и др.), а также получение, как нормализованной формы слова, так и всех его словоформ.

Основная проблема при разработке лексического и алгоритмического обеспечения компонентов морфологического процессора [2] состоит в хранении словаря большого объема и, соответственно, большого времени поиска в нем.

В последние десятилетия многими исследователями были разработаны системы морфологического анализа, синтеза и нормализации, как для русского языка [3–6], так и для многих других языков, использующие различные методы поиска в словарях. Так в [3] предложена эффективно реализуемая модель морфологии флективного естественного языка, в словарях которой используется поиск методом полного перебора, в [4] при вероятностном морфологическом анализе используется аналогичный алгоритм. Авторы работы [5] предлагают использовать при морфологическом анализе бинарный поиск в инверсионном списке основ.

Постановка задачи

Используемые в рассмотренных работах алгоритмы поиска не эффективны для предложенной автором в [2] структуры словаря, вследствие различных структур данных его составляющих. Таким образом, актуальна задача выбора наиболее эффективного алгоритма для каждого компонента словаря.

Анализ существующих методов поиска в словарях

Существующие в настоящее время методы поиска можно классифицировать следующим образом:

1. Статические и динамические.
2. Методы, использующие «истинные» или преобразованные ключи.

Наиболее распространенными являются:

- полный перебор всех терминов словаря, или последовательный поиск;
- метод расширения выборки, или метод спел-чекера;
- метод n-грам (триад);
- trie-деревья;
- триангуляционные деревья;
- метод поиска по бинарному дереву;
- поиск по "дереву Фибоначчи";
- метод экстраполяций.
- методы цифрового поиска.

Статический и динамический поиск. При использовании статического метода поиска словарь не изменяется, а при динамическом поиске значения словаря могут перестраиваться или может изменяться размерность словаря.

Модуль морфологического процессора для каждой лексической единицы выполняет поиск аффиксов в морфологическом словаре [2], который не изменяются во время поиска. Следовательно применяемый в данном случае алгоритм поиска обязательно будет статическим.

Методы «истинных» и преобразованных ключей. «Ключом» называют то значение аффикса, которое ищется. Поиск в словаре является поиском по преобразованным ключам, так как все аффиксы уже отсортированы в алфавитном порядке, то есть массив значений был изменен перед началом поиска.

Рассмотрим подробнее перечисленные выше методы.

Полный перебор. Если нет никакой дополнительной информации о разыскиваемых данных, то очевидный подход – простой последовательный пересмотр массива с увеличением шаг за шагом той его части, где желаемого элемента не обнаружено. Данный метод осуществляет последовательное считывание значений из словаря и их

последующее сравнение с искомым аффиксом. Условием окончания поиска является нахождение желаемого элемента или достижение конца массива без обнаружения совпадения. Если элемент найден, то он найден вместе с первым минимальным из возможных индексов, т.е. это первый из таких элементов. Если совпадения не было, то окончание цикла произойдет через N шагов. На каждом шаге необходимо увеличивать индекс и проверять сложное логическое условие (проверка на совпадение с ключом и проверка на достижение конца массива). Для ускорения поиска необходимо заменить условие на более простое. Это можно сделать, при условии, что совпадение обязательно произойдет. Ожидаемое число сравнений N/2. Основным достоинством этого метода является простота реализации. Метод прямого перебора оптимален для малых массивов. Если же объем словаря достаточно велик, то оптимальность поиска достигается предварительной сортировкой значений в словаре.

Метод расширения выборки. Суть данного метода заключается в следующем: строится множество всевозможных "ошибочных" слов, например, получающихся из исходного в результате одной операции редактирования, после чего построенные термины ищутся в словаре (на точное соответствие) [6]. Данный метод неприменим в нашем случае в виду ограниченного числа аффиксов во флективных языках.

Метод n-грам. Идея этого метода заключается в следующем: если строка v "похожа" на строку u, то у них должны быть какие-либо общие подстроки. Поэтому бывает целесообразно строить инвертированный файл словаря, в котором роль документов играют сами термины, а роль терминов - подстроки длины n, называемые также n-грамами [7]. Основной недостаток метода n-грам - большой размер файла.

Trie-деревья. В отличии от обычных сбалансированных деревьев, в trie-дереве все аффиксы, имеющие общее начало, располагаются в одном поддереве. Каждое ребро помечено некоторым аффиксом. Терминальным вершинам ("листьям") соответствуют слова списка.

Обычно trie-деревья используются для поиска по подстроке, но их можно использовать, и весьма эффективно, для поиска по сходству.

Триангуляционные деревья. Триангуляционные деревья позволяют индексировать множества произвольной структуры, при условии, что на них задана метрика [7]. Существует довольно много различных модификаций этого метода, но все они не слишком эффективны в случае текстового поиска и чаще используются для организации поиска в базе данных изображений или других сложных объектов.

Метод поиска по бинарному дереву. Алгоритм, основанный на знании того, что массив упорядочен. Основная идея – выбрать случайно некоторый элемент, например a_i, и сравнить его с искомым. Если он равен ключу, то поиск заканчивается. Если он меньше ключа, то считаем, что все элементы с индексами меньше или равными i можно исключить из поиска.

Если же он больше ключа, то исключаются все элементы с индексами больше и равные i. Выбор i совершенно произволен и корректность алгоритма от него не зависит. Однако на эффективность работы алгоритма этот выбор влияет. Для оптимизации алгоритма необходимо исключить на каждом шагу из дальнейшего поиска, каким бы ни был результат сравнения, как можно больше элементов. Оптимальным решение будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива. В нашем случае суть этого алгоритма может быть следующая. Словарь аффиксов сортируется в алфавитном порядке. Например, необходимо найти в словаре коней *дуб*. Берем корень из середины словаря, получаем *лес*. Сравнивая его с искомым конем, видим, что первый символ в корне *дуб* предшествует первому символу в корне *лес*. Значит, корень *дуб* находится точно в первой половине словаря. Отбрасываем вторую часть словаря, сужая массив поиска в два раза. Теперь берем корень из середины оставшегося словаря и так далее. Таким образом , при каждом сравнении , мы уменьшаем зону поиска в два раза. Отсюда и название метода - половинного деления или дихотомии. Скорость сходимости этого алгоритма пропорциональна Log_2N . Это означает буквально то, что не более, чем через Log_2N сравнений, мы либо найдем нужное значение, либо убедимся в его отсутствии.

Другое название этого алгоритма – «метод бинарного дерева» происходит из представления "пути" поиска в виде дерева (у которого каждая следующая ветвь разделяется на две, по одной из которых мы и движемся в дальнейшем) [8].

Поиск по "дереву Фибоначчи". Эффективность метода немного выше, чем у поиска по бинарному дереву, хотя так же пропорциональна Log(2)N.

В дереве Фибоначчи числа в дочерних узлах, отличаются от числа в родительском узле на одну и ту же величину, а именно на число Фибоначчи [9]. Суть метода в том, что сравнивая наше искомое значение с очередным значением в массиве , мы не делим пополам новую зону поиска , как в бинарном поиске, а отступаем от предыдущего значения, с которым сравнивали, в нужную сторону на число Фибоначчи.

Этот способ считается более эффективным, чем предыдущий, так как метод Фибоначчи включает в себя только такие арифметические операции, как сложение и вычитание. Нет необходимости в делении на 2, тем самым экономится процессорное время.

Метод интерполяций. Как и в предыдущих случаях словарь сортируется в алфавитном порядке. Пусть искомый аффикс *лес*, начинается на букву «Л». Открываем словарь немного ближе, чем на середине. Нам попалась буква «К», ясно, что искать надо в первой части словаря, а на сколько отступить? На половину? Но это больше, чем нам необходимо. Ведь нам не просто известно, в какой части массива искомое

значение, мы владеем еще и информацией о том, насколько далеко надо шагнуть. В этом и заключается суть рассматриваемого метода. В отличии от двух предыдущих, он не просто определяет зону нового поиска, но и оценивает величину нового шага.

Алгоритм имеет скорость сходимости большую, чем первые методы. Если при поиске по бинарному дереву за каждый шаг массив поиска уменьшался с N значений до N/2 , то при этом методе за каждый шаг зона поиска уменьшается с N значений до корня квадратного из N. Если К лежит между Kn и Km , то следующий шаг делаем от n на величину (n - m)*(K - Kn)/(Km - Kn). Можно показать, что интерполяционный поиск требует в среднем около Log_2 log_2N шагов. К сожалению эксперименты показали [10], что данный метод уменьшает количество сравнений не настолько, чтобы компенсировать возникающий дополнительный расход времени. Алгоритм эффективен лишь при весьма больших N.

Метод цифрового поиска. В этом методе ключ представляется в виде последовательности символов, принадлежащих рассматриваемому алфавиту, а структура хранения данных – это M-арное дерево, где M равно количеству символов в алфавите. Каждый узел уровня L является набором всех ключей, начинающихся с определенной последовательности L символов. Узел определяет разветвление на M путей в зависимости от (L+1)-го символа.

Основным достоинством этого метода является его высокая скорость – log_MN, где N – количество ключей – основ или постфиксов (в зависимости от рассматриваемого словаря), M – размер вектора указателей. Его основным недостатком является огромный объем занимаемой памяти.

Алгоритмы поиска для отдельных компонентов словаря

В виду ограниченности количества приставок и с учетом незначительного числа составляющих их символов (см. рис.1) целесообразно использовать в словаре приставок метод полного перебора.

бес	пере	пре	при	рас	

Рисунок 1 – Словарь приставок

Время поиска в этом методе сокращается при предварительной сортировке значений в словаре. Для выбора наиболее подходящего алгоритма сортировки словаря префиксов в алфавитном порядке рассмотрим подходящие методы сортировки [8] и проведем их сравнение.

1) Сортировка включением. Одним из наиболее простых и естественных методов внутренней сортировки является сортировка с простыми включениями. Идея алгоритма очень проста. Пусть имеется

массив ключей a[1], a[2], ..., a[n]. Для каждого элемента массива, начиная со второго, производится сравнение с элементами с меньшим индексом (элемент a[i] последовательно сравнивается с элементами a[i-1], a[i-2] ...) и до тех пор, пока для очередного элемента a[j] выполняется соотношение a[j] > a[i], a[i] и a[j] меняются местами. Если удается встретить такой элемент a[j], что a[j] <= a[i], или если достигнута нижняя граница массива, производится переход к обработке элемента a[i+1] (пока не будет достигнута верхняя граница массива). Если длина нашего массива равна n, нам нужно пройтись по n - 1 элементам. Каждый раз нам может понадобиться сдвинуть n - 1 других элементов, получая алгоритм с временем работы $O(n^2)$.

2) Дальнейшим развитием метода сортировки с включениями является сортировка методом Шелла, называемая по-другому сортировкой включениями с уменьшающимся расстоянием. Эффективность метода Шелла объясняется тем, что сдвигаемые элементы быстро попадают на нужные места. Мы не будем описывать алгоритм в общем виде, а ограничимся случаем, когда число элементов в сортируемом массиве является степенью числа 2. Для массива с 2n элементами алгоритм работает следующим образом. На первой фазе производится сортировка включением всех пар элементов массива, расстояние между которыми есть 2(n-1). На второй фазе производится сортировка включением элементов полученного массива, расстояние между которыми есть 2(n-2). И так далее, пока мы не дойдем до фазы с расстоянием между элементами, равным единице, и не выполним завершающую сортировку с включениями.

3) Обменная сортировка. Простая обменная сортировка (в просторечии называемая "методом пузырька") для массива a[1], a[2], ..., a[n] работает следующим образом. Начиная с конца массива сравниваются два соседних элемента (a[n] и a[n-1]). Если выполняется условие a[n-1] > a[n], то значения элементов меняются местами. Процесс продолжается для a[n-1] и a[n-2] и т.д., пока не будет произведено сравнение a[2] и a[1]. Понятно, что после этого на месте a[1] окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, но последними сравниваются a[3] и a[2]. И так далее. На последнем шаге будут сравниваться только текущие значения a[n] и a[n-1]. Понятна аналогия с пузырьком, поскольку наименьшие элементы (самые "легкие") постепенно "всплывают" к верхней границе массива.

4) Сортировка выбором. При сортировке массива a[1], a[2], ..., a[n] методом простого выбора среди всех элементов находится элемент с наименьшим значением a[i], и a[1] и a[i] обмениваются значениями. Затем этот процесс повторяется для получаемых подмассивов a[2], a[3], ..., a[n], ... a[j], a[j+1], ..., a[n] до тех пор, пока мы не дойдем до подмассива a[n], содержащего к этому моменту наибольшее значение.

5) Сортировка разделением (Quicksort). Этот метод является развитием метода простого обмена и настолько эффективен, что его стали называть "методом быстрой сортировки - Quicksort". Основная идея алгоритма состоит в том, что случайным образом выбирается некоторый элемент массива x, после чего массив просматривается слева, пока не встретится элемент a[i] такой, что a[i] > x, а затем массив просматривается справа, пока не встретится элемент a[j] такой, что a[j] < x. Эти два элемента меняются местами, и процесс просмотра, сравнения и обмена продолжается, пока мы не дойдем до элемента x. В результате массив окажется разбитым на две части - левую, в которой значения ключей будут меньше x, и правую со значениями ключей, большими x. Далее процесс рекурсивно продолжается для левой и правой частей массива до тех пор, пока каждая часть не будет содержать в точности один элемент. Понятно, что как обычно, рекурсию можно заменить итерациями, если запоминать соответствующие индексы массива.

Проведем сравнение алгоритма поиска прямым перебором при различных способах сортировки.

Есть несколько факторов, влияющих на выбор метода сортировки в каждой конкретной ситуации.

1) Время сортировки - основной параметр, характеризующий быстродействие алгоритма.

2) Память - ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не будет учитываться место, которое занимает исходный массив и независящие от входной последовательности затраты, например, на хранение кода программы.

3) Устойчивость - устойчивая сортировка не меняет взаимного расположения равных элементов. Такое свойство может быть очень полезным, если они состоят из нескольких полей, в нашем же случае элементы словаря префиксов состоят из одного поля.

В таблице 1 представлено количество элементов массива. Проведем сравнение по времени различных методов сортировки при поиске прямым перебором.

Таблица 1 – Количество элементов

№	1	2	3	4	5	6	7
Кол-во элементов	2	4	8	16	32	64	128

Для простых методов сортировки (включением, прямой выбор, прямой обмен) можно получить формулы, по которым вычисляется

минимальное, максимальное и среднее число сравнений ключей (C) и пересылок элементов массива (M).

Таблица 2 - Число сравнений ключей и пересылок элементов массива

Метод	Min	Avg	Max
Прямое включение	C = n-1 M = 2^x(n-1)	(n^2 + n - 2)/4 (n^2 - 9n - 10)/4	(n^2 -n)/2 - 1 (n^2 -3n - 4)/2
Прямой выбор	C = (n^2 - n)/2 M = 3^x(n-1)	(n^2 - n)/2 n^x(ln n + 0.57)	(n^2 - n)/2 $n^2/4$ + 3^x(n-1)
Прямой обмен	C = (n^2 - n)/2 M = 0	(n^2 - n)/2 (n^2 - n)^{x}0.75	(n^2 - n)/2 (n^2 - n)^{x}1.5

Результаты сравнения приведены на рисунке 2.

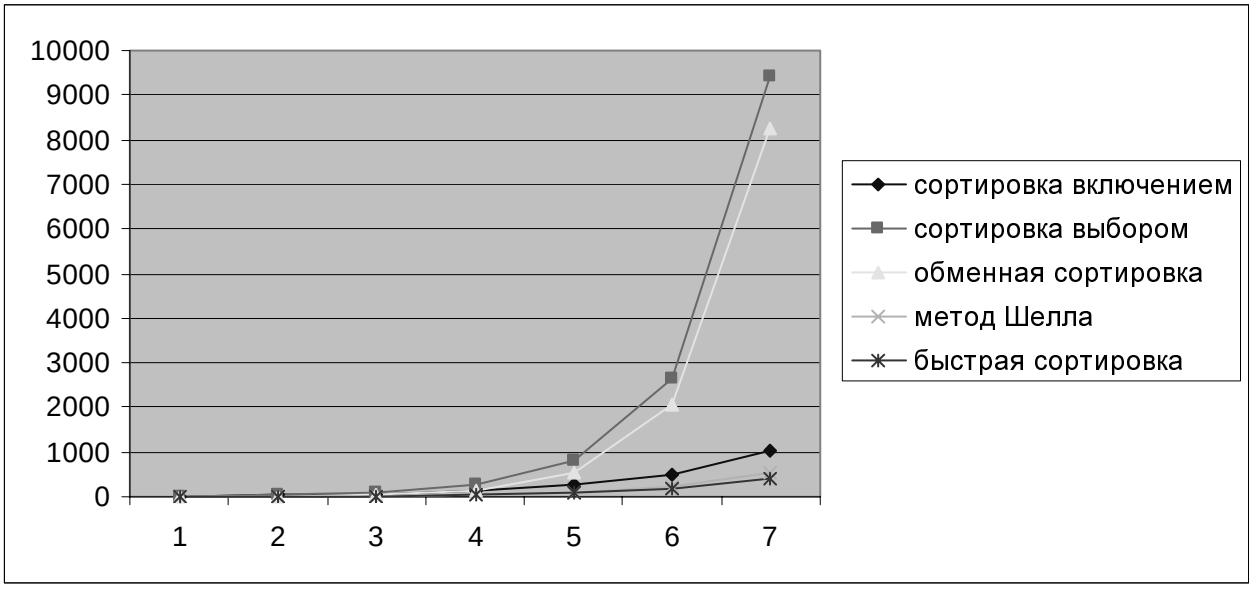

Рисунок 2 – Сравнение методов сортировки при поиске прямым перебором

На основе проведенных экспериментов можно сделать вывод, что наиболее оптимальным по быстродействию методом сортировки префиксов является метод «быстрой» сортировки.

Учитывая тот факт, что группа корневых и суффиксальных аффиксов является наиболее многочисленной и в них чаще всего употребляется чередование, предлагается использовать для словаря корней и суффиксов метод Trie-деревьев.

Рассмотрим применение данного метода. Пусть список аффиксов содержит основы *снег* (снег), *снеж* (снежок), *кот* (кот), *кошк* (кошка). Соответствующее trie-дерево изображено на рисунке 3.

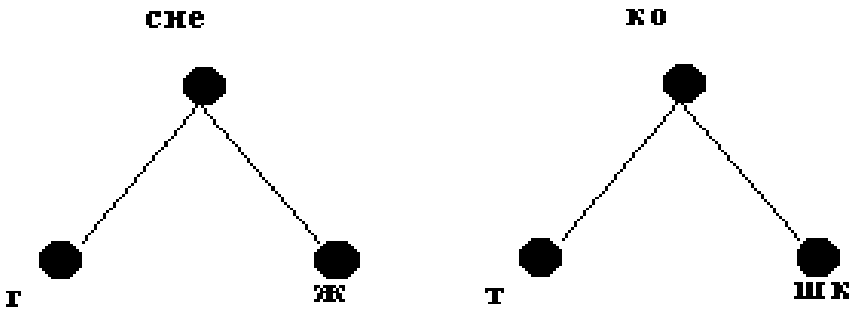

Рисунок 3 - Trie-дерево для корневых аффиксов

Построим аналогичное Trie-дерево для словаря, содержащего следующие суффиксы: *чик* (водопроводчик), *щик* (каменщик).

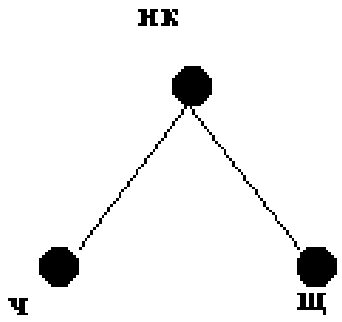

Рисунок 4- Trie-дерево для суффиксальных аффиксов

Теперь рассмотрим алгоритм выделения морфем в лексеме и их последующий поиск в словарях.

Обобщенный алгоритм поиска состоит из следующих шагов.

Шаг 1. Производится поиск в словаре окончаний: от искомого слова «отрезается» по одному символу, пока полученная на текущем шаге комбинация символов не будет найдена в словаре окончаний. Например, определим окончание для слова «вычислений». На первом шаге «отрезается символ «й», производится поиск данного символа в словаре окончаний, результат поиска положителен.

Шаг 2. Следующим шагом является разбиение основы на приставку, корень и суффикс. Данный алгоритм предложен для лексем, в состав которых входит не более одного суффикса и префикса. Алгоритмы определения приставки и суффикса аналогичны. Единственное отличие

состоит в направлении «отсечения». Продолжим морфологический разбор для слова «вычислений». На данном этапе будем обрабатывать основу «вычислен». Аналогично шагу 1 «отрезаем» символ «н», проводим его поиск в словаре суффиксов. Одновременно проводим поиск оставшейся части основы в словаре корней. Если совпадения и в словаре корней, и в словаре суффиксов обнаружены, то полученное разбиение соответствуют суффиксу и корню. Иначе продолжаем «отрезать» по одному символу и проводить аналогичный поиск в словаре корней и суффиксов (необходимо учитывать тот факт, что максимальная длина предполагаемого суффикса не должна превышать 6 символов, а минимальная длина предполагаемого корня – 1 символ). Если указанные ограничения достигнуты, а корень и суффикс так и не определены, то выдвигается гипотеза о наличии перед корнем приставки.

Шаг 3. При наличии гипотезы о существовании в искомом слове приставки будем отделять по одному символу с начала и с конца слова, образовывая все возможные сочетания приставки, корня, суффикса и проверяя их наличие в словарях. На примере полученной на первом шаге основы «вычислен», рассмотрим этот процесс:

1. *в-ычисле-ни*: результат поиска в словаре суффиксов положителен, результаты поиска в словарях приставок и корней отрицательны;
2. *в-ычисл-ени*: результат поиска в словаре суффиксов положителен, результаты поиска в словарях приставок и корней отрицательны;
3. *вы-числе-ни*: результат поиска в словаре суффиксов и приставок положителен, результат поиска в словаре корней отрицателен;
4. *вы-числ-ени*: результаты поиска во всех словарях положительны, выделены приставка (вы), корень (числ), суффикс (ени).

Выводы

Повышение эффективности поиска в словарях позволяет повысить быстродействие подсистемы морфологического анализа. В работе проведен анализ наиболее распространенных методов поиска, позволяющий выбрать наиболее эффективные из них для каждого конкретного случая. Обоснованно применение выбранных методов поиска для различных составляющих словаря морфологического анализатора. Предложен обобщенный алгоритм поиска составляющих слово морфем в словаре. В дальнейшем планируется использовать предложенную структуру словаря и методы поиска в нем в модуле морфоанализа интеллектуальной ИПС для САПР.

Литература

1. Егошина А.А. Языковые и алгоритмические аспекты построения морфологических процессоров для интеллектуального поиска в полнотекстовых базах данных – VI международная конференция

«Интеллектуальный анализ информации ИАИ-2006».: Киев, 16-19 мая 2006 г.: Сб. тр./Рос.ассоц.искусств.интеллекта и др.; Под ред. Т.А. Таран.-К.:Просвіта, 2006. – 334 с.: ил

2. Егошина А.А., Об одном способе построения статического словаря морфологического процессора // Искуственный интеллект. – 2007. – № 2.

3. Гельбух А.Ф., Минимизация количества обращений к диску при словарном морфологическом анализе // Научно-техническая информация, серия 2. — М.: ВИНИТИ, 1991, N 6.

4. Ермаков А.Е., Плешко В.В., Компьютерная морфология в контексте анализа связного текста. Компьютерная лингвистика и интеллектуальные технологии: труды Международной конференции Диалог’2004. – Москва, Наука, 2004

5. Сегалович И., Маслов М., Русский морфологический анализ и синтез с генерацией моделей словоизменения для не описанных в словаре слов. ЭТАП-2. М.,1989.

6. Ёлкин С.В., Бетин В.Н., Жигарев А.Е, Простаков О.В, Хачукаев Э.М. Разработка семантического анализа текстов при автореферировании // Вестник ВИНИТИ НТИ. 2001. сер 2, N12. С. 18-21

7. Ricardo Baeza-Yates, Gonzalo Navarro, Fast Approximate Matching in a Dictionary. In 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98), Sta. Cruz de la Sierra, Bolivia, September 1998. IEEE CS Press.

8. Кнут Д., "Искусство программирования для ЭВМ", 1978г, издательство "МИР", том 3 "Сортировка и поиск".

9. Воробьев В.В., "Числа Фибоначчи. Популярные лекции по математике, выпуск №6. 1969г. Издательство "Наука"

10.Альсведе Р., Вегенер И., "Задачи поиска", 1982г, Издательство "Мир"