Санкт-Петербургский государственный университет

Факультет прикладной математики - процессов управления
Кафедра технологии программирования

Мозговой
Максим
Владимирович

Информационный поиск:

выявление и использование семантических зависимостей в предложении

Зав. кафедрой,
кандидат физ.-мат. наук,
доцент Сергеев С.Л.

Научный руководитель,
доктор физ.-мат. наук,
профессор Тузов В.А.

Рецензент,
доктор физ.-мат. наук,
профессор Братчиков И.Л.

Санкт-Петербург
2003

Оглавление

Введение

Современные системы информационного поиска (searching engines), к сожалению, не отличаются особой интеллектуальностью. Фактически, всё, что они могут сделать — это найти документы, которые «некоторым образом» соответствуют введённому запросу. Если вам надо найти документ, содержащий известную цитату или получить список книжных онлайн-магазинов, большинство поисковых систем сумеют помочь. Но если запрос нельзя сформулировать столь прямолинейно, вас ждёт жестокое разочарование (точнее, сформулировать запрос можно, но поисковая система его не поймёт правильно). К примеру, попробуйте найти «Войну и мир» в Яндексе [1]. В ответ на запрос «Лев Толстой. Война и мир» вы получите огромное количество ссылок на документы, *содержащие* эти слова. Нетрудно догадаться, что само произведение отнюдь не переполнено фразами «война и мир», поэтому собственно текст романа (являющийся наиболее релевантным документом, конечно же) окажется далеко не в фаворитах. Самыми же релевантными документами будут страницы, содержащие списки рефератов, ибо рефератов по «Войне и миру» много, и название каждого из них почти всегда содержит эти слова. Отмечу, что система Google [2] более умна — она сразу находит текст самого романа. Думаю, что «интеллектуальность» её в данном случае обусловлена лишь тем, что она придаёт большее значение *заголовкам* интернет-страниц, чем Яндекс. Логично, что документ, озаглавленный «Война и мир», с большой вероятностью содержит текст произведения. Тем не менее, это лишь эвристика, пригодная разве что для подобных случаев. Можно придумать много других примеров, когда качество современных поисковых систем оказывается абсолютно неудовлетворительным. Один из наиболее ярких — поиск документа,

содержащего *неизвестное* слово. Например, вас интересует, как теперь называется бывшая Пушкинская улица в Москве, или какая кривая используется при проектировании поворотов на железной дороге. Скорее всего, найти эту информацию будет очень трудно. Насчёт Пушкинской улицы, может быть, ещё повезёт, если найдётся документ, в котором будет написано что-то вроде: «…наш адрес: ул. Большая Дмитровка (бывшая Пушкинская), дом такой-то» (например, на сайте какой-нибудь организации), а насчёт железнодорожных путей — вряд ли.

Несомненно, все мы ждём, когда же, наконец, появятся по-настоящему интеллектуальные системы, которые будут действительно способны понять, что мы хотим найти. К сожалению, до их создания ещё очень далеко (я смею лишь надеяться, что доживу до этого славного времени), а поиск информации — вопрос насущный. Что же остаётся делать разработчикам поисковых систем сегодня и сейчас? Например, добавлять в программу возможности, которые могут быть полезны при поиске хотя бы в некоторых случаях. Типичный «джентльменский набор» поисковой системы может быть, к примеру, таким:

- логические операции при поиске (и/или/не);
- поиск точной фразы (цитаты);
- возможность указать слова, которые обязательно должны содержаться в документе;
- поиск изображений, mp3-файлов;
- поиск файлов с известным именем;
- использование расстояния между словами.

Моя работа описывает ещё одну полезную функцию, которую можно включить в этот список — *поиск связанных слов*.

Что было раньше?

Поиск связанных слов идеологически наиболее близок последнему пункту «джентльменского набора» — использованию расстояния между словами. Когда поисковая система индексирует некоторый документ, каждому его слову присваивается порядковый номер. Расстояние между двумя словами — это просто разница их порядковых номеров [3]. Если в документе встречается сочетание красная шапочка, расстояние между словами красная и шапочка равно единице, а расстояние между словами шапочка и красная — минус единице. Требуемое расстояние можно ввести прямо в строке поиска: к примеру, если в Яндексе ввести запрос «поставщики /2 кофе», то найдутся документы, в которых содержится и слово «поставщики», и слово «кофе», а расстояние между этими словами будет не больше двух. Таким образом, релевантными окажутся документы, содержащие фразы вида «поставщики колумбийского кофе», «поставщики кофе из Колумбии» и т.п.

Основное назначение этого инструмента, конечно же — поиск слов, связанных по смыслу. Если нас интересует чёрный кофе, простой поиск по этим двум словам выдаст множество документов, содержащих оба слова, хотя и никак не связанных между собой. С другой стороны, поиск по точной фразе «чёрный кофе» окажется слишком ограниченным: сочетания вроде «чёрный молотый кофе» или «чёрный бразильский кофе» не будут найдены. Поиск с расстоянием — разумная альтернатива. Например, в ответ на запрос «чёрный /2 кофе» можно получить и просто «чёрный кофе», и «чёрный молотый кофе», и «чёрный бразильский кофе». Однако документ, содержащий сочетание «чёрный молотый бразильский

кофе» уже не будет считаться релевантным. Можно, конечно, попробовать ещё увеличить расстояние («чёрный /3 кофе»), но при этом увеличится доля нерелевантных документов, содержащих фразы вроде «чёрный пакет с кофе».

Если смотреть в корень проблемы, всё дело в том, что расстояние между словами — плохой критерий их взаимной зависимости. Два слова могут стоять рядом в предложении и при этом никак не относиться друг к другу; с другой стороны, порою отнюдь не соседние слова оказываются очень близко связанными по смыслу. Расстояние используется не потому, что оно даёт хорошие результаты; просто более точных критериев не нашли.

Что будет теперь?

К счастью, времена меняются. Пусть мы пока ещё не в силах изобрести по-настоящему умную поисковую машину, но реализовать поиск связанных слов уже можно гораздо более передовым способом, по крайней мере, на голову превосходящим примитивный подход с расстояниями. Разработке такого способа и посвящена эта работа.

Архитектура системы

Описываемая здесь система состоит из двух основных частей: простого поисковика, построенного на базе модели векторного пространства и модуля, в котором определяются связанные слова. Замечу, что система уже реализована — приведённые описания (намеренно избавленные от слишком глубоких технических подробностей) лишь отражают грани реально существующего продукта.

Сейчас мы познакомимся с отдельными концепциями, лежащими в основе поисковой системы. Сначала обсудим модель векторного пространства — что это такое и как она может быть использована в

задаче информационного поиска. Затем разберёмся, каким образом «движок» (ядро поисковика, в котором нет ничего, кроме прямолинейной реализации принципов модели векторного пространства) можно расширить новыми возможностями, используя дополнительные модули.

Модель векторного пространства

Рассмотрим сначала более простую задачу. Допустим, имеется некоторая коллекция документов и документ-запрос (обычный запрос, который вы вводите в любой поисковой системе тоже можно считать коротким документом, состоящим всего из нескольких слов). Требуется найти множество документов из коллекции, *релевантных* (то есть каким-то образом соответствующих) данному запросу и отсортировать их по убыванию релевантности.

Одна из самых простых методик решения этой проблемы, использующихся в наше время, называется *моделью векторного пространства* (Vector Space Model, VSM [4, 5]). Обозначим через T общее количество разных слов во всех документах коллекции (иными словами, размер используемого словаря). Ограничив словарь, можно присвоить каждому слову некоторый порядковый номер и в дальнейшем ссылаться на слово, используя этот номер. Обозначим теперь через $w_{i,d}$ так называемый *вес* слова i в документе d. О том, как его определить, речь пойдёт позже. Зная веса слов, можно «описать» любой документ d вектором ($w_{1,d}$, $w_{2,d}$, …, $w_{T,d}$) (иначе говоря, каждому документу единственным образом сопоставляется некоторый вектор). Обратите внимание, что в описании любого документа присутствуют веса всех слов, которые только есть в словаре. Как правило, методики определения веса слова приписывают отсутствующему в документе слову вес, равный нулю. Отсюда вывод: поскольку любой конкретный документ, скорее всего, не содержит и

половины слов из словаря, большинство компонентов такого вектора будут равны нулю (поэтому для его хранения обычно используют специальные структуры данных вроде разрежённого массива).

Конечно, сама идея описания документа подобным образом очень спорна (а ещё более спорны способы вычисления веса каждого слова), но за неимением значительно лучших идей люди до сих пор используют эту подкупающую своей простотой модель 1975 года.

Договорившись о том, что любому документу соответствует вектор, можно перенести все операции с документами в хорошо известное и изученное векторное поле. Например, векторы (в отличие от документов) легко сравнивать, используя их скалярное произведение. Определим функцию подобия sim(a, b) двух векторов следующим образом:

$$sim(a,b) = \frac{(a,b)}{|a| \times |b|} = \frac{\sum_{i=1}^{T} w_{i,a} \times w_{i,b}}{\sqrt{\sum_{i=1}^{T} w_{i,a}^2} \times \sqrt{\sum_{i=1}^{T} w_{i,b}^2}}$$

Поскольку все векторы, с которыми мы работаем, имеют одинаковую размерность, их можно перемножать без опасений. Скалярное произведение векторов равно произведению длин векторов на косинус угла между ними, поэтому, разделив скалярное произведение в формуле на произведение длин, мы получим косинус, используемый, в итоге в качестве меры релевантности. Для очень похожих документов значение формулы будет близким к единице, для непохожих — к нулю. Таким образом, вычислив функцию подобия для запроса (который, как уже было сказано, является документом) и всех остальных документов коллекции, мы получим искомые релевантности. Если релевантность некоторого документа окажется меньше некоторого заданного порогового значения τ, его можно

считать вообще нерелевантным. Остальные документы ранжируются по убыванию значения меры релевантности и возвращаются пользователю.

Приступим теперь к задаче вычисления веса слова. Классическая теория предписывает использовать формулу

$$w_{i,d} = \frac{freq_{i,d}}{\max_{l=0...T}(freq_{l,d})} \times \log\frac{N}{n_i}$$

Здесь $freq_{i,d}$ — это количество вхождений слова i в документ d. Разумно предположить, что часто встречающиеся слова имеют в документе больший «вес», чем слова редкие. Количество вхождений обязательно надо нормализовать, разделив на максимальное значение этой величины для документа, поскольку интересно не количество само по себе, а его относительная величина (одно дело, если некоторое слово встречается сто раз в документе, содержащем миллион слов, и совсем другое — если документ содержит лишь тысячу слов). Величина $\log(N / n_i)$, где N — общее количество документов коллекции, а n_i — количество документов, содержащих слово i, называется *обратной частотой документа* (inverse document frequency). Она призвана уменьшить вес тех слов, которые слишком часто встречаются в коллекции. Действительно, если вся коллекция посвящена компьютерам, бессмысленно придавать большое значение слову «компьютер», поскольку почти все документы его содержат.

Модель векторного пространства можно использовать и для решения задачи рубрикации — объединения схожих документов в классы («работа», «компьютеры», «досуг» и т. п.) и определения класса для только что поступившего документа. Поскольку функцию подобия можно вычислить для любых двух документов, сравнительно несложно разбить всю коллекцию на классы.

Коснусь ещё двух технологий, позволяющих улучшить качество работы модели векторного пространства — стемминга и уничтожения стоп-слов.

Стемминг

Большинство слов могут стоять в разных формах. Хорошая поисковая система должна рассматривать разные вариации слова как единое целое, как единственное вхождение в словарь. Например, если запрос содержит слово «компьютер», поисковая система должна быть способна найти все документы, содержащие слова «компьютеры», «компьютеров» и тому подобные. Чтобы добиться такого поведения, проще всего выбрать для каждого слова некоторую «нормальную форму» и приводить все словоформы к этой нормальной форме. В качестве нормальной формы часто используют основу (стем) слова. Основу можно определить автоматически при помощи специальных алгоритмов. Например, для английского языка существует широко известный и достаточно качественный алгоритм Портера [5, 6]. Мне неизвестны подобные алгоритмы для русского, но простой орфографический словарь, при помощи которого можно определить начальную форму слова по любой словоформе, с успехом может его заменить. Алгоритмы вроде портеровского работают быстро и не требуют больших ресурсов; однако использование словаря позволяет добиться лучшего качества (к примеру, алгоритм Портера ошибочно приводит слова animation и animal к одной основе anim).

Таким образом, на практике поисковые системы обычно имеют дело не с реальными словами документа, а с их нормальными формами или основами.

Уничтожение стоп-слов

Многие слова, содержащиеся в документах, не представляют никакой пользы для поиска. Фактически, они даже мешают правильному поиску, искажая реальное содержимое документа и увеличивая затраты памяти и времени. Такие слова обычно называют *стоп-словами*. К стоп-словам относятся слова, не имеющие своего собственного значения («а», «но», «следовательно», «или», …) и слова, которые можно встретить с большой вероятностью в самых разнообразных документах (числа, названия дней недели и месяцев). На практике проще всего создать список таких стоп-слов и полностью игнорировать их во время поиска. Вообще говоря, составление списка стоп-слов — вопрос достаточно тонкий. Реальные списки могут сильно варьироваться от системы к системе; если разработчики одного поисковика сочли некоторое слово малозначимым, разработчики другого могут придерживаться противоположного мнения.

Практические аспекты: кластеры и профили

На практике почти все системы используют некоторые идеи, хотя и не обязательные с точки зрения теории, но весьма полезные (в первую очередь с точки зрения производительности). Сейчас мы познакомимся с некоторыми из них.

Как правило, реальные поисковики работают с большими коллекциями. Я бы даже сказал, с очень большими коллекциями. По этой причине просто невозможно провести полную процедуру сравнения запроса с каждым документом (это затянется на часы, если не на дни). Стандартный способ решения этой проблемы называется *кластеризацией*. Я уже упоминал о том, что модель векторного пространства можно использовать для создания рубрикатора. Так вот, при кластеризации используется именно эта идея: каждый документ

попадает в некоторую категорию (называемую *кластером*). Категории либо строятся автоматически, либо задаются создателями системы. Вовсе необязательно, чтобы категории были осмысленными; достаточно, чтобы их количество было разумным, и чтобы наполнялись они более-менее равномерно (если в одной категории лежит половина коллекции — верный признак, что не всё в порядке). В каждом кластере некоторым образом выбирается или строится *центроид* (как именно — отдельная тема для разговора), то есть документ, который наилучшим образом соответствует всему кластеру (иными словами, этот документ наиболее релевантен кластеру в целом). Теперь при поиске вовсе необязательно сравнивать запрос с каждым документом коллекции — достаточно выбрать кластер, соответствующий запросу (путём сравнения последнего с центроидом каждого кластера), а затем уже производить поиск только в выбранном кластере, а не во всей коллекции.

Поисковые системы также никогда не ищут информацию в настоящих документах. Вместо них используются так называемые *профили* — записи, содержащие необходимую информацию о реальных документах. Профили создаются для того, чтобы избежать ресурсоёмких операций извлечения этой информации из документов при каждой процедуре поиска. Профиль каждого документа может содержать, к примеру, такие данные:

- уникальный идентификатор (номер) записи;
- ссылка на реальный документ коллекции;
- список всех слов документа;
- список позиций, в которых встречается каждое слово в реальном документе;
- максимальная частота встречаемости слова в данном документе.

Потребуется также создать и *профиль коллекции*, содержащий данные:

- список всех слов коллекции;
- список профилей документов, в которых содержится данное слово;
- частота данного слова в данном документе.

Если в коллекцию поступают новые документы, следует произвести операцию *индексирования*, то есть создания профилей новых документов и обновления профиля коллекции.

Поскольку все профили хранятся в двоичном виде, невозможно привести реальный пример их организации. Однако я могу продемонстрировать основные идеи, используя упрощённое текстовое представление.

Фрагмент базы данных профилей документов:

```
123<"MyCollection/icq/protocol.txt">
(
    icq <5>10,17 <130>15 <160>12
    format <95>6
    ...
    representation <51>7 <1201>12

    max_freq 21
)
124<"MyCollection/icq/userguide.txt">
(
    configure <10>17 <140>25
    system <19>6 <25>10,16 <190>11
    ...
    proxy <151>6 <321>16 <503>11

    max_freq 15
)
...
125<"MyCollection/java/applets.txt">
(
    sun <1>11 <101>8
```

```
    applet <15>6,20 <25>13 <170>21 <201>9
    ...
    browser <125>8 <202>11 <345>3

    max_freq 33
)
```

Рассмотрим запись №124. Она соответствует документу коллекции `MyCollection/icq/userguide.txt`. Этот документ содержит слова `configure`, `system` и так далее. Слово `system` встречается на шестой позиции в 19-й строке, на десятой и шестнадцатой позициях в 25-й строке и на одиннадцатой позиции в 190-й строке реального документа. Максимальная частота встречаемости слова равна 15 — это означает, что некоторое слово встречается в документе 15 раз.

Фрагмент профиля коллекции:

```
abandon <15>11 <49>20 <101>5 <120>3
adjust <90>2 <93>1 <103>10
area <1>4 <102>2 <111>4
...
yacc <99>3 <101>1
```

Из этого фрагмента можно заключить, к примеру, что слово `yacc` может быть найдено в записи №99 три раза, а в записи №101 всего один раз.

Дополнительные возможности

Из приведённого описания видно, что модель векторного пространства не поддерживает возможности вроде поиска с использованием булевых функций, поиска цитат или связанных слов. Всё, что она может — это каким-то образом сравнить два документа и определить, насколько они похожи друг на друга.

Тем не менее, добавить эти возможности довольно просто. Для каждого подобного действия создаётся отдельный модуль, который

изменяет содержимое коллекции перед тем, как её будет анализировать главная программа. К примеру, пусть задан запрос «Керниган & Ричи». Модуль, анализирующий логические операции, отберёт из коллекции лишь те документы, которые содержат оба слова, и вызовет главную программу для этой изменённой коллекции (при этом запрос изменится на «Керниган Ричи»).

Возвращаясь к системе, описываемой здесь, я должен заметить, что поиск связанных слов осуществляется по такому же принципу. Сначала специальный модуль, предназначенный только для поиска связанных слов, отбирает документы, содержащие заданные зависимости. Затем вызывается главная программа, ранжирующая по релевантности отобранные документы (см. рис. 1).

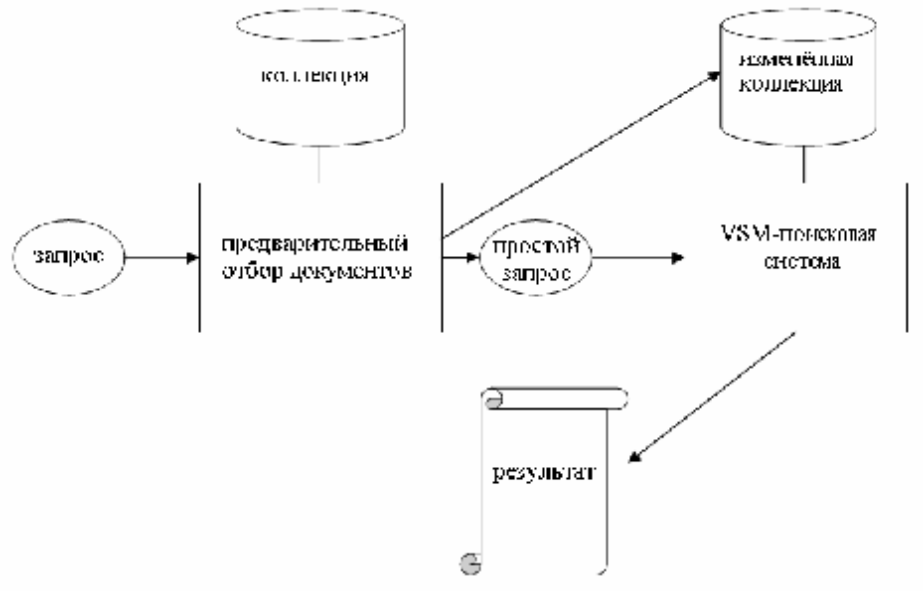

Рис. 1. Интеграция дополнительных модулей в систему

Поиск связей: теоретическое обоснование

Займёмся теперь модулем, который осуществляет поиск связанных слов.

Главным теоретическим основанием для моих идей служит работа Виталия Алексеевича Тузова [7]. Процитирую некоторые её тезисы, необходимые для дальнейшего понимания текста:

«Тезис 1. *Язык есть алгебраическая система* ***{f1,f2,...,fn,M)****, где* ***fi*** *– функции,* ***M*** *– структура данных языка.*

Этот тезис дает принципиальный ответ на вопрос, как устроен язык, и утверждает существование единой универсальной грамматики для всех языков – как искусственных, так и естественных.

Тезис 2. *Адекватная грамматика приписывает каждому предложению структурное описание в виде суперпозиции функций.*

Если какая-либо грамматика приписывает какому-то предложению нечто, отличное от суперпозиции (древовидную структуру, графовидную или нечто подобное), то такая грамматика не может быть адекватной естественному языку.

Тезис 3. *Грамматика конкретного языка есть прямая конкретизация универсальной грамматики.*

Если два языка семантически эквивалентны, то их отличие выражается лишь в именах (названиях) функций и в формах вызова их аргументов.

Тезис 5. *Грамматика неразрывно связана с семантикой языка и представляет собой семантический словарь.*

Каждое слово должно быть описано некоторой совокупностью семантических формул. Множество таких описаний есть семантический словарь. Если словарь не содержит описания какого-то слова, то можно считать, что это слово вообще не существует. Степень абстрактности формулы может зависеть от многих причин,

но чем менее абстрактна формула, тем точнее компьютер может реагировать на слово, связанное с ней.

Следствие 1. *Лучше очень абстрактно описать слово, чем не описывать его вовсе.*

Следствие 2. *Синтаксическая структура предложения является адекватным отражением его семантической структуры*».

Основной упор в работе В. Тузова делается на построение семантического словаря, то есть словаря, описывающего смысл того или иного слова русского языка. Другая важная тема книги — морфологический анализ слов (определение части речи, её рода и т. п.) Имея семантический словарь и морфологический анализатор, можно построить так называемый *семантический анализатор*, назначение которого — «вычисление» смысла предложения. Разумеется, и построение семантического словаря, и конструирование морфологического и семантического анализаторов — очень сложные, объёмные и важные темы, которые заслуживают самого пристального внимания. Тем не менее, для правильного определения связанных слов (давайте вернёмся к основной теме работы) совсем необязательно изучать *внутреннее* устройство этих изощрённых алгоритмов. Готовые к использованию морфологический и семантический анализаторы уже существуют (в реализации В. Тузова), и сейчас главное для нас — это знать, что именно они делают, и как мы можем воспользоваться результатами их работы. Разумеется, такой сложный инструмент как семантический анализатор не может гарантировать корректность разбора во всех случаях без исключений, однако работает на практике он совсем неплохо; более того, его качество постоянно улучшается.

Реально семантический анализатор претворяет в жизнь приведённые выше тезисы, переводя обычные предложения русского языка в «структурное описание в виде суперпозиции функций» (тезис 2). Рассмотрим, к примеру, результат разбора предложения «Кто лишает себя всяческих иллюзий, тот останется нагим»:

```
=====================================================
 Кто лишает себя всяческих иллюзий, тот останется нагим.
.....................................................
лишает<X001.004>
  (@Им Кто<X000.001>\<X000.002><+$Им_svod0_8.0+>,
   @Род себя<X002.001><+$Мест_svod0_9.0\@Вин+>,
   @Род иллюзий<X004.001><+ShR2_svod1_14.0+>
    (@Род всяческих<X003.001><+$Какой3_svod0_10.0+>),
   останется<X007.002><+SLodm7_svod1_15.0+>
    (@Им тот<X006.001><+$Им_svod0_11.0+>),
   @Тв нагим<X008.001><+прилТв1_svpr>_20.0+>)
 .
=====================================================
Кто
  <X000.001> КТО {Мест. Одуш $1~@ОНЪ$17@Им $1~@ОНА$17@Им
$1~@ОНО$17@Им $1~@Я$17@Им $1~@ТЫ$17@Им} $1(Z0:s> НЕЧТО$1,Z1: !?p)
  s: <X000.001> КТО Copul_o1(НЕЧТО$1~!%1,Z1)
  <X000.002> КТО {Мест. Одуш $124~@ОНЪ$17@Им $124~@ОНА$17@Им
$124~@ОНО$17@Им $124~@Я$17@Им $124~@ТЫ$17@Им} N%~ЖИВОЙ$124(Z0:s>
ЖИВОЙ$124,Z1: !?p)
  s: <X000.002> КТО Copul_o1(ЖИВОЙ$124~!%1,Z1)
лишает
  <X001.004> ЛИШАТЬ {Глаг} N%~ПОТЕРЯ$151531231(Z1:
!ОНЪ$17\!ОНА$17\!ОНО$17,Z2: ЧЕЛОВЕК$1241~!Вин,Z3: !Род)
  s: <X001.004> ЛИШАТЬ Caus(Z1,FinHab(Z2,Z3))
            <X001.004>Z1@ОНЪ$17 => <X000.002>
            <X001.004>Z1@ОНА$17 => <X000.001>
            <X001.004>Z1@ОНО$17 => <X000.002>
            <X001.004>Z2@Вин => <X002.001>
            <X001.004>Z3@Род => <X002.001>
            <X001.004>Z3@Род => <X004.001>
            <X001.004>@Тв => <X008.001>
себя
  <X002.001> СЕБЯ {Мест. @Род @Вин} $17()
  s: <X002.001> СЕБЯ ()
```

```
всяческих

  <X003.001> ВСЯЧЕСКИЙ {Прил. $1~@ОНИ$17@Род}
N%~ВСЕВОЗМОЖНЫЙ$1100/11(Z0:a> НЕЧТО$1)

  s: <X003.001> ВСЯЧЕСКИЙ Mult^e(РАЗНЫЙ$1100/11(НЕЧТО$1~!%1))

иллюзий

  <X004.001> ИЛЛЮЗИЯ {Сущ. Жен Неодуш $131331~@ОНИ$17@Род}
$131331(Z1: !Род,Z2: !вПред\!сТв\!?p)

  s: <X004.001> ИЛЛЮЗИЯ (Z1,Z2)

            <X004.001>*@$Какой3 => <X003.001>

тот

  <X006.001> ТОТ {Сущ. Муж Неодуш $1100/16~@ОНЪ$17@Им}
$1100/16(Z0:a> ЭТОТ$1100/16,Z1: !сТв)

  s: <X006.001> ТОТ Ne(ЭТОТ$1100/16~!%1)(Z1)

останется

  <X007.002> ОСТАТЬСЯ {Глаг} N%~ОСТАТОК$111034(PerfFut Z1:
!ОНЪ$17\!ОНА$17\!ОНО$17,Z2: !Ото)

  s: <X007.002> ОСТАТЬСЯ PerfIncep_Copul(Z1,ОСТАТОК$111034(Z2))

            <X007.002>Z1@ОНЪ$17 => <X006.001>

нагим

  <X008.001> НАГОЙ {Прил. МжСр $1241~@ОНЪ$17@Тв $1241~@ОНО$17@Тв}
N%~ОДЕЖДА$12136(Z0:a> ЧЕЛОВЕК$1241)

  s: <X008.001> НАГОЙ NeHab_a1(ЧЕЛОВЕК$1241~!%1,ОДЕЖДА$12136)
```

Лишь на первый взгляд понять эту распечатку совершенно невозможно; на самом же деле в ней закодирована важная и вполне извлекаемая информация.

Первый блок данных (от ряда точек до ряда знаков «равно») содержит предложение, записанное в виде той самой суперпозиции функций, о которой уже говорилось. Помимо слов в записи присутствует информация о *классах* (каждое слово по Тузову принадлежит некоторому смысловому классу вроде МЕБЕЛЬ, ЧЕЛОВЕК или ОДЕЖДА), падежах и разные отладочные данные. Далее идут блоки, дающие развёрнутую информацию о каждом слове по отдельности (тоже не лишённую отладочных данных). Из этих блоков можно получить справку об описании слова в семантическом словаре, его классе, принадлежности к той или иной части речи и конкретном значении в данном предложении. Поскольку значения слов для нас не

важны (мы интересуемся только связями), рассмотрим подробнее первый блок.

При помощи специальной программы (написанной мною) можно избавиться от классов, падежей и отладочной информации, оставив лишь скобочную форму записи предложения. В данном случае результат будет таким:

```
лишает(Кто,себя,иллюзий(всяческих),останется(тот),нагим)
```

Теперь можно увидеть суть тезиса 2 непосредственно: как именно любое предложение может быть представлено в виде суперпозиции слов-функций.

Выявление связей в формализованном предложении

Теперь на арену выходит следствие 2 («синтаксическая структура предложения является адекватным отражением его семантической структуры»). Если некоторые слова связаны между собой по смыслу, эти зависимости можно определить исходя из вида скобочной формы записи предложения, которую мы получили в результате работы семантического анализатора. Более наглядно эта форма может быть представлена в виде дерева (см. рис. 2). Каждое слово в предложении будет являться узлом дерева, а потомками узла станут аргументы этого слова (помните, что любое слово является ещё и функцией от некоторого числа других слов-аргументов).

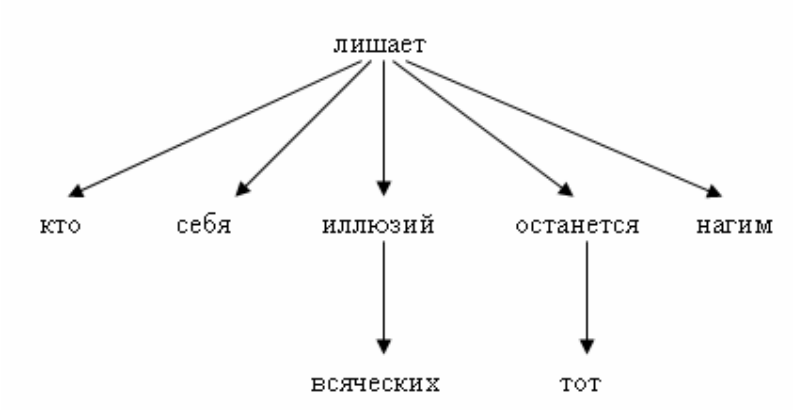

Рис. 2. Представление предложения в виде дерева

В соответствии со следствием 2 это дерево отражает реальные смысловые связи между словами. Я выделяю два типа таких связей: *связь-свойство* и *связь-отношение*.

Связи первого типа возникают, когда одно слово является свойством (или *как бы* является свойством) второго. Например, именно такая связь существует между словами чёрный и кофе в предложении «я пью чёрный кофе» или между словами Большой и театр в предложении «я иду в Большой театр». В сочетании из трёх слов «чёрный бразильский кофе» любые два слова связаны подобной зависимостью. На уровне дерева предложения это означает выполнения двух требований:

1. оба слова находятся на одной ветви дерева;
2. участок дерева между словами имеет вид *списка* (иными словами, каждый узел, который встречается в дереве на пути от одного слова до другого, имеет ровно один дочерний[1]).

Рассмотрим, к примеру, дерево предложения «я купил чёрный бразильский кофе» (см. рис. 3).

[1] На узел, соответствующий слову, стоящему в дереве ниже по иерархии, ограничений не накладывается.

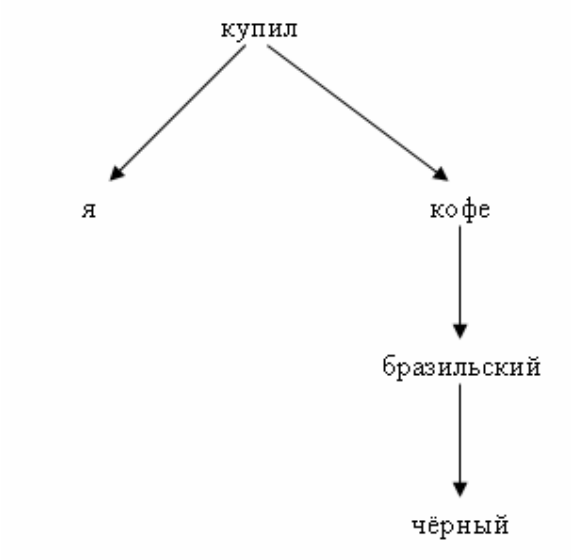

Рис. 3. Дерево фразы «я купил чёрный бразильский кофе»

Слова чёрный и кофе, чёрный и бразильский, бразильский и кофе связаны по смыслу, причём связью первого типа. Слово купил не находится в зависимости первого типа с этими словами, потому что у соответствующего ему узла два потомка — я и кофе. Поскольку слово я не находится на одной ветви с кофе, оно тоже не связано со словами кофе, бразильский и чёрный связью-свойством.

Связь-отношение возникает между словами, когда имеет место связь-свойство между каждым из них и некоторым третьим словом. В приведённом примере связь такого типа существует между словами я и кофе, я и бразильский, я и чёрный. Можно сказать, что слова я и кофе находятся в отношении «купил».

Рассмотрим другие примеры возникновения связей в предложениях:

1. «Маша любит Сашу» (рис. 4).

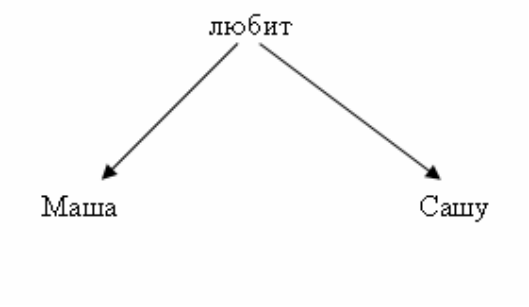

Рис. 4. Дерево фразы «Маша любит Сашу»

Эта фраза — простейший пример возникновения связи-отношения (в данном случае между словами Маша и Сашу). Скобочная форма записи предложения хорошо отражает этот факт:

```
любит(Маша,Сашу).
```

Других связей в предложении нет.

2. «Большой театр находится в Москве» (рис. 5).

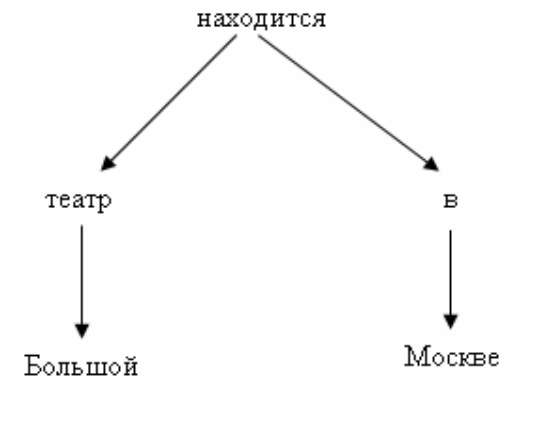

Рис. 5. Дерево фразы «Большой театр находится в Москве»

Слова Большой и театр, а также в и Москве связаны зависимостью первого типа. Каждое из слов Большой и театр также связано с каждым из слов в и Москве зависимостью второго типа.

3. «Я купил кофе в чёрном пакете» (рис. 6).

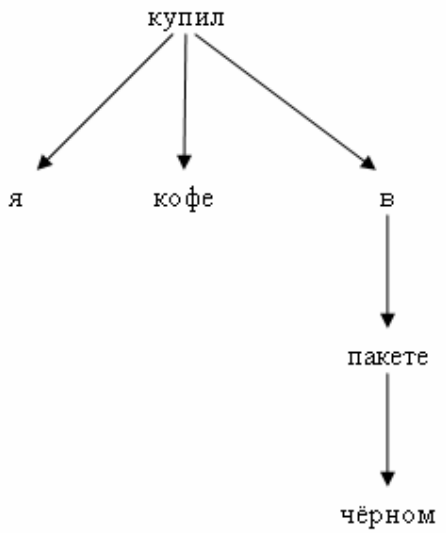

Рис. 6. Дерево фразы «я купил кофе в чёрном пакете»

Это предложение — очень хороший пример того, как семантический анализатор корректно распределяет не связанные между собой (хотя и стоящие почти рядом) первым типом зависимости слова кофе и чёрном по разным ветвям дерева. Здесь слово чёрном связано первым типом зависимости лишь со словами пакете и в. Со словом кофе оно тоже связано, но тип зависимости здесь — второй.

Практическая реализация модуля

Напомню, что моя главная цель — создание модуля, который бы расширял возможности поисковой системы нахождением связанных по смыслу слов. Мы уже обсудили, как устроена связка «система — дополнительные модули» и на чём основан принцип поиска связанных слов. Пора объединить эти идеи в одно целое.

Архитектура модуля изображена на рис. 7.

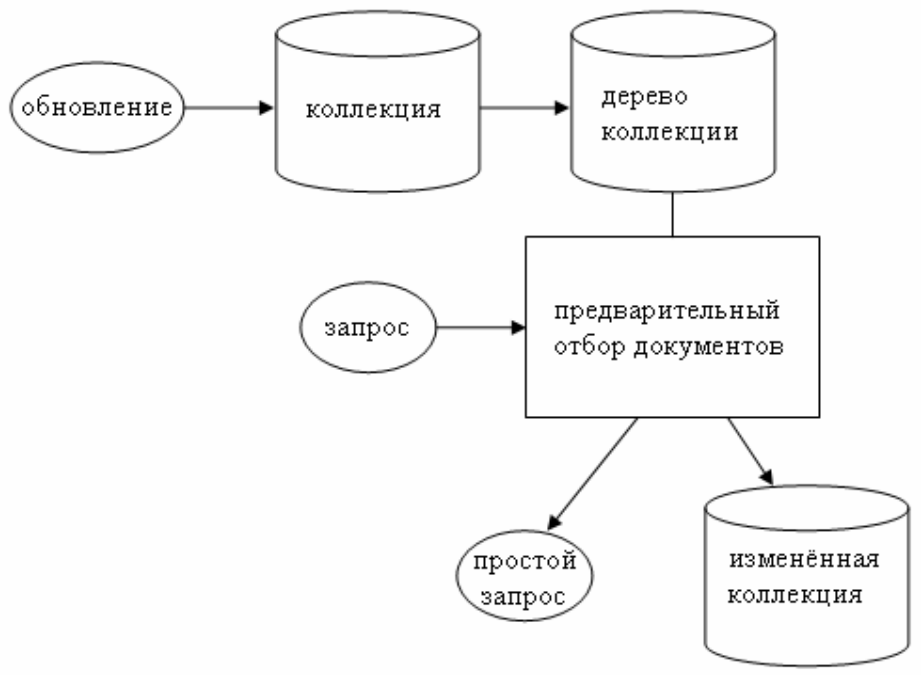

Рис. 7. Архитектура модуля поиска связанных слов

Как и сама поисковая система, модуль имеет две точки входа. Первая предназначена собственно для поиска связанных слов, а вторая — для обновления дерева коллекции. Теоретически дерево коллекции можно полностью строить «на лету», на практике же решение плохое, поскольку операция эта требует значительного времени.

Рассмотрим подробнее обе операции (обновления и поиска) по отдельности.

Операция обновления строит для каждого документа коллекции соответствующее ему дерево. Мы уже сталкивались с деревьями, которые соответствуют отдельным предложениям. Эти деревья можно объединить под общим корнем и тем самым получить дерево всего документа (см. рис. 8).

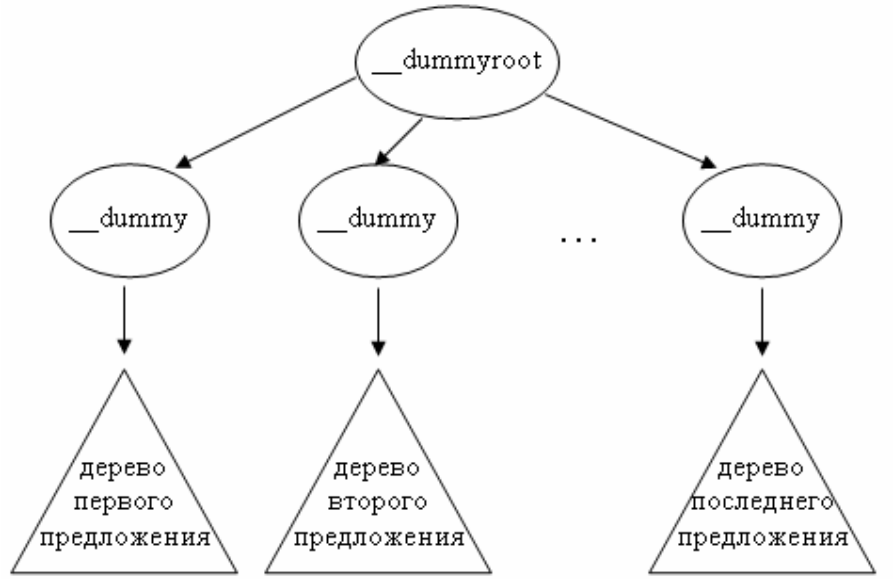

Рис. 8. Дерево документа

Для дерева каждого предложения выделяется дополнительный элемент __dummy, наличие которого гарантирует, что в дереве документа не возникнет новых, реально не существующих логических связей (по моей классификации) из-за того, что все предложения объединены под одним корнем. Если напрямую соединить деревья предложений с элементом __dummyroot, то окажется, что все слова, являющиеся корнями соответствующих предложений, находятся между собой в зависимости второго типа (а в действительности это не так).

Для построения такого дерева каждый документ коллекции обрабатывается в четыре этапа.

На первом этапе работает семантический анализатор, который строит для данного документа структурное описание в виде суперпозиции функций.

На втором этапе действует процедура очистки полученного описания от ненужной нам информации — сведений о падежах в предложении, о каждом слове по отдельности и отладочных данных.

На третьем этапе ко всем словам описания, сгенерированного на прошлом шаге, применяется процедура стемминга.

На четвёртом этапе результаты предыдущего шага используются для построения дерева документа. Готовое дерево сохраняется в специальном файле.

Рассмотрим теперь операцию поиска. Поскольку в документах я выделяю два разных типа логической связи, имеет смысл дать пользователю возможность выбора — какой именно тип он ищет в данном конкретном случае. Я избрал для этого «скобочный» синтаксис. Чтобы указать, что два слова должны находиться между собой в зависимости первого типа, их следует заключить в круглые скобки. Зависимость второго типа задаётся при помощи квадратных скобок. Допустим, запрос состоит из слов Маша Саша чёрный кофе. Если мне требуется, чтобы слова Маша и Саша находились между собой в зависимости-отношении, а чёрный и кофе — в зависимости-свойстве, следует переписать запрос так: [Маша Саша] (чёрный кофе).

Получив на входе запрос, модуль поиска анализирует его, находя пары слов, заключённые в круглые или квадратные скобки. Остальные слова его не интересуют. Найденные пары сохраняются в специальном списке.

Затем программа по очереди проверяет все деревья, соответствующие документам из коллекции. «Проверка» дерева заключается в поиске в нём каждой зависимости из списка. По умолчанию я решил считать документ релевантным[2], если он содержит хотя бы одну связь из перечисленных в запросе (то есть используется логическое ИЛИ);

[2] В данном случае под «релевантным» я подразумеваю документ, который войдёт в коллекцию, подлежащую анализу во время работы основной поисковой системы.

однако несложно изменить это поведение на логическое И или вообще сделать его настраиваемым. Исходный запрос без скобок образует простой запрос, а документы, сочтённые модулем поиска связей релевантными — изменённую коллекцию (см. рис. 7, иллюстрирующий архитектуру модуля). Процедура, которая ищет зависимости в дереве документа, является прямолинейной интерпретацией определений связей обоих типов на языке программирования.

Итоги

1. Можно не делать ничего в ожидании революции (или хотя бы прорыва) в области информационного поиска, а можно существенно улучшить характеристики существующих систем, дополняя последние новыми модулями.
2. Сами модули могут быть сколь угодно сложными, однако добавить такой модуль в готовую разумно спроектированную систему довольно просто.
3. Хороший пример такого модуля — предлагаемая здесь программа поиска связанных слов в коллекции документов, доведённая до практического воплощения.
4. Определение связанных слов — лишь простейший пример использования семантического анализатора; более творческий подход открывает очень заманчивые перспективы (вплоть до построения систем, «понимающих» тексты на русском языке).

Заметки на полях

С технической точки зрения в программе я не вижу проблем. Её можно развивать и дальше — например, запрограммировать поиск зависимостей не между двумя, а между тремя и более словами,

увеличить скорость работы (используя более эффективные алгоритмы), добавить новые булевы функции (чтобы работали запросы вида [Маша Саша] И НЕ (чёрный кофе)). Главные вопросы возникают в той области, где представления компьютера о зависимостях слов пересекаются с представлениями человека.

Допустим, я нашёл два типа зависимостей, научился находить их в дереве предложения, более или менее формально описал их с точки зрения человека и с точки зрения компьютера. Вопрос в том, насколько эти категории естественны для человека? Действительно ли мы оперируем подобными понятиями или они трудны для понимания и не совсем соответствуют природе нашего мышления? Думаю, это открытый вопрос.

Учительница русского языка просит ученика найти в предложении слова, связанные между собой по смыслу. Что она при этом имеет в виду? Я сомневаюсь, что вы сможете получить от неё внятный ответ. Все мы люди, и все мы устроены схожим образом. Поэтому каждый из нас представляет, что такое хорошо и что такое плохо, что есть любовь и красота. Пусть наши представления об этих вещах разнятся, но всё равно у любого человека есть какой-то собственный образ, ассоциирующийся с этими словами; тот самый образ, который делает их для нас отнюдь не пустыми звуками. Именно благодаря тому, что все мы люди, мы и можем понимать (с большей или меньшей степенью успеха) друг друга. Компьютеры устроены совершенно иначе. Они оперируют абсолютно другими категориями. То, что человек делает с лёгкостью, для компьютера может быть запредельно сложно и наоборот. Компьютер не понимает, что значит «связаны по смыслу», ему надо объяснить, расписать это понятие по полочкам. Проблема же (я бы даже сказал, трагикомизм) ситуации заключается в том, что мы сами толком не знаем, о чём говорим. Что есть красота?

Это же очевидно, каждый из нас представляет, что такое красота! Что значит «слова связаны по смыслу»? Ну, как же, любой может это прочувствовать (хотя каждый при этом по-своему). Ситуация становится ещё более забавной, когда мы из мира эмоций и чувств переходим в сферу, казалось бы, холодного разума. Ни один гроссмейстер не сумел объяснить, как именно он думает, когда играет в шахматы[3]. Очевидно, что человек отнюдь не перебирает миллионы вариантов ходов за секунду, когда принимает решение, куда теперь передвинуть фигуру (это удел компьютера). Мы «видим», мы «чувствуем», мы «ощущаем» понятия, «напрашивающиеся» решения задач, сокрытый в словах смысл. Вполне возможно, что человек на самом деле и мыслит в категориях связей-отношений и связей-свойств, но не отдаёт себе в этом отчёта. А может быть, это вовсе не так.

Когда в языкознании заходит речь о связях между словами, под этим обычно подразумевается возникновение ситуаций, имеющих большее отношение к синтаксису, чем к смыслу. К примеру, в русском языке существует определённая иерархия связей (см. рис. 9).

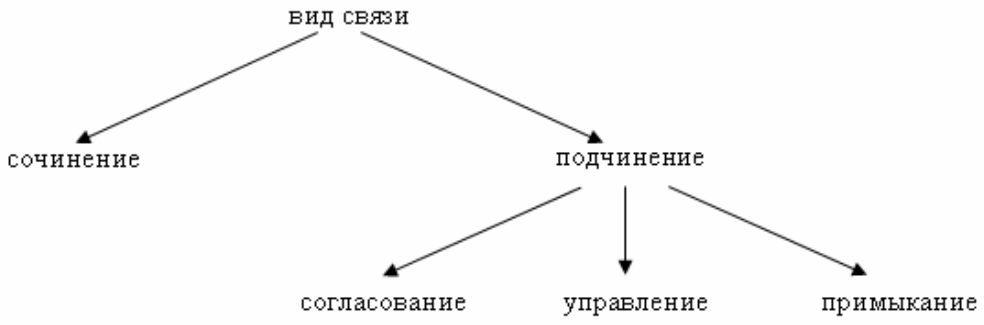

Рис. 9. Связи в русском языке

[3] В этом смысле очень показательна история Михаила Ботвинника, который, будучи выдающимся шахматистом и прекрасным программистом, так и не сумел создать приличной шахматной программы, как ни пытался.

В книге [8] они определяются следующим образом (выделение подчёркиванием — моё): «При сочинении в связь вступают синтаксически равноправные, независимые друг от друга элементы (члены предложения), например: *книга и тетрадь (лежат на столе)*; *(читаю) книги, газеты, журналы*. При подчинении в связь вступают синтаксически неравноправные элементы (один зависит от другого), например: *читать книгу, совет друга*.

Подчинение имеет три разновидности синтаксических связей: согласование, управление и примыкание.

Согласование — это такой вид подчинительной связи, при котором зависимое слово уподобляется в своей форме господствующему слову, например: *важный вопрос, главная улица, новые дома*. <…>

Управление — это такой вид подчинительной связи, при котором зависимое слово ставится в определенной падежной форме (без предлога или с предлогом), обусловленной лексико-грамматическим значением господствующего слова, например: *читать письмо, интересоваться искусством, любовь к родине*. <…>

Примыкание — это такой вид подчинительной связи, при котором зависимость подчиненного слова выражается лексически, порядком слов и интонацией. Примыкают неизменяемые знаменательные слова (наречие, инфинитив, деепричастие), например: *тихо шептать, предложить войти, говорить улыбаясь*».

Как видно из цитаты, во-первых, лингвисты предлагают нам самим определить, какие именно слова связаны между собой, как будто бы это просто и очевидно[4]. Если данный вопрос решён, уже нетрудно определить, с каким именно типом связи мы имеем дело. Понятно, что

[4] …а это и есть просто и очевидно для человека, но никак не для компьютера!

если «один [элемент] зависит от другого», то перед нами подчинительная связь; однако никто не предлагает нам алгоритма определения того, зависит ли один элемент от другого или нет; мы должны как-то «догадаться». Во-вторых, и это главное, подобная классификация имеет дело с теми или иными явлениями конкретного языка (в данном случае русского), почти не пытаясь проникнуть в смысловую суть фразы. Стоит лишь обратиться к английскому языку, так сразу же половина элементов этой классификации исчезает: в английском нет ни родов, ни падежей, ни окончаний (в нашем понимании этих терминов).

Логические связи, конечно же, не имеют никакого отношения к падежам или другим явлениям какого-либо конкретного языка; логические связи находятся как бы над языком, не завися от него. В моей классификации в предложениях “I’m drinking black coffee” и “Joe loves Emily” существуют те же самые связи, что и в их русских эквивалентах — (black coffee) и [Joe Emily].

Всё вышесказанное, тем не менее, вовсе не перечёркивает положения моей работы. Моя работа не является *неверной*, она всего лишь *спорна*, как и абсолютно все другие работы, связанные с анализом человеческих представлений о таких тонких вещах как «смысл», «связь» или, скажем, «знание». Я лишь предлагаю возможную (но не единственно возможную) классификацию; feci quod potui faciant meliora potentes. Самый лучший способ проверить её качество — провести практические исследования, спросить пользователей, насколько естественной кажется им работа с этой системой (я не в счёт — опыт позволяет мне почти всегда моментально видеть все «интересные» связи в предложении).

Закончить мне бы хотелось итоговым списком вопросов, которые ещё ждут своего часа (если угодно, можно считать их разрешение «планами на будущее»):

1. Корректна ли моя классификация? Действительно ли связь-свойство и связь-отношение в моём определении этих понятий соответствуют реальным концепциям, которым они призваны соответствовать?
2. Существуют ли другие типы логических связей в предложении и насколько они интересны на практике?
3. Насколько удобно человеку оперировать этими типами связей в процессе поиска? Если я, к примеру, хочу найти два связанных по смыслу слова, сколько секунд (минут, часов) у меня уйдёт, чтобы сообразить, какой именно зависимостью они связаны?
4. Теоретическая интересность подобных исследований очевидна. А насколько востребована возможность указывать связанные слова при поиске в коллекции документов на практике?

Литература и ссылки

[1] Поисковая система Яндекс (www.yandex.ru)

[2] Поисковая система Google (www.google.com)

[3] Детальное описание языка запросов поисковой системы Яндекс (www.yandex.ru/ya_detail.html)

[4] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. Communications of the ACM, 18:613-620, 1975.

[5] Baeza-Yates R., Ribeiro-Neto B. Modern Information Retrieval. Addison Wesley, New York, 1999.

[6] Porter's algorithm description on Martin Porter’s homepage (http://www.tartarus.org/~martin/PorterStemmer)

[7] Тузов В.А. Компьютерная лингвистика. Опыт построения компьютерных словарей.

[8] Валгина Н.С., Розенталь Д.Э., Фомина М.И. Современный русский язык. Москва, "Логос", 2001.

Приложения

1. Программа, очищающая выходной файл семантического анализатора от неиспользуемых данных (остаётся лишь скобочная форма).

```
#include <stdio.h>
#include <string.h>
//-------------------------------------------------------------
FILE *infile, *outfile;
//-------------------------------------------------------------
void ProcessSentence()
{
 char str[500];

 for(;;)
 {
     if(!strncmp(fgets(str, 500, infile), ".........", 10))
         break;
     str[strlen(str) - 1] = 0;
     fprintf(outfile, "%s", str);
 }
 fprintf(outfile, "\n");
}
//-------------------------------------------------------------
void ProcessFunction()
{
 char str[500];
 for(;;)
 {
     if(!strncmp(fgets(str, 500, infile), "==========", 10))
         break;
     char *p = str;
     while(*p == ' ')        // skip leading spaces
         p++;
     while(*p == '(')           // print '(' symbol if present
         fputc(*p++, outfile);
     if(*p == '@')           // skip @some_word if present
     {
```

```
        while(*p != ' ')
            p++;
        p++;
    }

    // if the next character belongs to a word
    if(*p != ')' && *p != ',' && *p != '.' && *p != '!' &&
       *p != '?')
        while(*p != '<' && *p != '+' && *p != ',' &&
              *p != ')' && *p != '\n')
        {
            if(*p != '\"')
                fputc(*p, outfile);    // output the main word
            p++;
        }
    while(*p)
    {
        if(*p == ')')
            fputc(')', outfile);
        if(*p == ',')
            fputc(',', outfile);
        p++;
    }
 }

 fprintf(outfile, "\n\n"); // end of function reached
}
//-------------------------------------------------------------
int main(int argc, char* argv[])
{
 if(argc != 3)
 {
     printf("Usage: extractor <infile> <outfile>\n");
     return 1;
 }

 char str[500];
 infile = fopen(argv[1], "rt");
 outfile = fopen(argv[2], "wt");

 for(;;)
```

```
 {
     do
         if(fgets(str, 500, infile) == NULL)
             goto skip;                        // end of file
     while(strncmp(str, "==========", 10));
     ProcessSentence();
     ProcessFunction();
 }

skip:
 fcloseall();
 return 0;
}
//-------------------------------------------------------------
```

2. Избранные фрагменты модуля поиска связанных слов — процесс поиска связей в дереве.

```
//-------------------------------------------------------------
struct QueryElement
{
    string word1, word2;
    bool is_child;

    QueryElement(const string& a, const string& b, bool c) :
                            word1(a), word2(b), is_child(c) {}
};

bool InPredicateRel(const string& word, Node *node)
{
    if(word == node->value)
        return true;
    if(node->children.size() != 1)
        return false;
    return InPredicateRel(word, &node->children.front());
}

bool FoundInTree(const QueryElement& qe, Node *root)
{
```

```
    if(root->value == qe.word1 || root->value == qe.word2)
    {
        string foundstr =
                 root->value == qe.word1 ? qe.word1 : qe.word2;
        string tofoundstr =
                 root->value == qe.word1 ? qe.word2 : qe.word1;

        if(qe.is_child)   // ищем предикативную связь
        {
            if(InPredicateRel(tofoundstr, root))
                return true;
        }
        else             // ищем отношение
        {
            if(root->parent)
                for(list<Node>::iterator p =
                        root->parent->children.begin();
                        p != root->parent->children.end(); p++)
                    if(InPredicateRel(tofoundstr, &(*p)))
                        return true;
        }
    }

    // в текущем узле ничего не найдено
    for(list<Node>::iterator p = root->children.begin();
                                p != root->children.end(); p++)
        if(FoundInTree(qe, &(*p)))
            return true;

    return false;
}

void Search()  // поиск деревьев, удовлетворяющих запросу
{
    CollDir = "processed\\";
    FillFilesList();

    FILE *f = fopen("query.txt", "rb");
    char buffer[500];

    list<string> terms;
```

```
do
{
    fscanf(f, "%s", buffer);
    terms.push_back(buffer);
}
while(!feof(f));

list<QueryElement> pairs;
for(list<string>::iterator p = terms.begin();
                                    p != terms.end(); p++)
{
    list<string>::iterator next = p;
    next++;

    if((*p)[0] == '[')
        pairs.push_back(QueryElement(p->substr(1,
        p->length() - 1), next->substr(0, next->length() –
                                              1), false));
    else if((*p)[0] == '(')
        pairs.push_back(QueryElement(p->substr(1,
        p->length() - 1), next->substr(0, next->length() –
                                               1), true));
}

FILE *gf = fopen("goodfiles.txt", "wt");
for(list<string>::iterator p = files.begin();
                                    p != files.end(); p++)
{
    LoadTreeFromFile(*p);
    for(list<QueryElement>::iterator qp = pairs.begin();
                                  qp != pairs.end(); qp++)
        if(FoundInTree(*qp, root))
        {
            fprintf(gf, "%s\n", p->c_str());   // OR op.
            break;
        }

    delete root;
    // искать в файле (т.е. дереве) зависимости из pairs
    // если найдена хоть одна, добавить файл в список
    // "хороших" файлов
```

```
    }

    fclose(gf);
}
//---------------------------------------------------------------
```