ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего профессионального образования

НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

На правах рукописи

ИЛЬВОВСКИЙ

Дмитрий Алексеевич

МЕТОДЫ И АЛГОРИТМЫ ОБРАБОТКИ ТЕКСТОВЫХ ДАННЫХ НА ОСНОВЕ ГРАФОВЫХ ДИСКУРСИВНЫХ МОДЕЛЕЙ

Специальность 05.13.18

Математическое моделирование, численные методы и комплексы программ

Диссертация на соискание ученой степени кандидата технических наук

Научный руководитель

доктор физико-математических наук

С.О. Кузнецов

Москва, 2017

Оглавление

Введение

Актуальность работы. Моделирование языковых процессов порождает значительное количество открытых проблем, связанных с развитием соответствующего математического аппарата, созданием и реализацией эффективных алгоритмов и комплексов программ. К настоящему моменту разработано значительное количество хорошо развитых моделей текста, позволяющих (помимо представления текста) вычислять сходство между текстами: «мешок слов», n-граммы, синтаксические деревья разбора и т.д. Среди исследователей, внесших значительный вклад в разработку и применение этих моделей в прикладных задачах (для английского языка), можно отметить C.Manning, H.Schutze, D.Jurafsky, S.Abney, M.Collins, A.Moschitti и многих других. Подавляющее большинство реализованных на практике моделей не полностью учитывает структурные особенности текста, ограничиваясь либо частотными характеристиками слов и n-грамм, либо синтаксическими связями внутри отдельных предложений. Эти модели не позволяют работать с текстом на уровне фрагментов, состоящих из нескольких связанных предложений – абзацев. К другому классу моделей относятся многочисленные лингвистические теории, в той или иной степени учитывающих дискурсивные связи между предложениями. Здесь можно отметить работы таких исследователей как W.Mann, D.Marcu, J.Searle, I.Mel’cuk, H.Kamp, M.Recaesens, D.Jurafsky и многих других. Однако эти модели обладают уже другим недостатком: они носят по большей части теоретический характер, не имеют полного математического или алгоритмического описания и не могут напрямую быть использованы для решения прикладных задач. В то же время учет дискурсивных связей внутри абзаца является критическим фактором в

таких важных задачах, как поиск по сложным и редким запросам, кластеризация поисковой выдачи по сложным запросам, классификация текстовых описаний. Всё это делает применение существующих моделей текста затруднительным и требует разработки новой модели, которая была бы предназначена для решения перечисленных задач, одновременно обладала достаточной теоретической базой и была реализуема на практике.

Необходимость интеграции в модель сложных структурных описаний и применения модели для задач кластеризации делает актуальным применение методов, позволяющих работать со структурным сходством и использовать эффективные приближения описаний. Методы теории решеток замкнутых описаний предоставляют удобный и эффективный математический аппарат для построения моделей в решении целого ряда важных научных и прикладных задач, в число которых входит и работа с текстами. Эта теория позволяет осуществлять концептуальную кластеризацию и находить сходство произвольного множества объектов (в частности, текстов). Включенный в теорию аппарат проекций позволяет эффективно работать с приближенными описаниями, в той или иной мере учитывающими основные свойства структуры и понижающими вычислительную и временную сложность обработки этих описаний.

Объект исследований – математические модели текстов на естественном языке. **Предмет исследований** – модели текстов на естественном языке, предназначенные для поиска, классификации и кластеризации текстовых данных.

Целью диссертационного исследования является разработка моделей и методов представления и обработки текстов на естественном языке, учитывающих синтаксическую и дискурсивную

структуру текстового абзаца и ориентированных на применение в задачах поиска, классификации и кластеризации текстовых данных.

К **задачам исследования** относятся:

- Разработка структурной модели текстов на естественном языке, ориентированной на поиск, классификацию и кластеризацию текстов и использующей синтаксические и дискурсивные связи внутри текста;
- Применение построенной модели в задаче поиска сходства текстов с целью улучшения релевантности поиска по сложным запросам;
- Применение построенной модели в задаче классификации текстов с целью повышения качества существующих методов за счет использования дискурсивной информации;
- Построение на основе разработанной модели таксономического представления текстовых документов с использованием решеток замкнутых структурных описаний и применение представления в задаче кластеризации текстов;
- Разработка математической модели и метода для определения связи «та же сущность» в построенных на основе текстовых данных формальных описаниях и эффективная алгоритмическая реализация данной модели.
- Реализация разработанных моделей, методов и алгоритмов в виде программного комплекса.

К **методам**, использовавшимся в исследовании, относятся:

- Методы построения и анализа решёток замкнутых описаний;
- Методы фильтрации решеток понятий на основе индексов качества моделей;

- Методы построения проекций моделей на узорных структурах;
- Методы построения структурных моделей для текстовых данных;
- Методы построения синтаксических и дискурсивных моделей текста;
- Методы порождения моделей, основанных на графовом представлении.

Научная новизна. В диссертации получен ряд новых научных результатов, которые **выносятся на защиту**:

1. Разработана графовая модель текстов, использующая и обобщающая структурное синтактико-дискурсивное представление текстового абзаца (чащу разбора). Новизна модели заключается в совместном использовании синтаксических деревьев разбора и дискурсивных связей для представления текстовых абзацев на английском языке. Модель ориентирована на применение в задачах поиска, классификации и кластеризации текстов и позволяет описывать сходство текстов в терминах обобщения их структурных графовых и древесных описаний.
2. Предложенная модель применена в задаче поиска ответов по сложным запросам. Разработан численный метод, использующий разработанную модель. Применение метода позволяет улучшить качество поиска и устранить недостатки существующих моделей благодаря применению впервые введенной в работе операции структурного синтактико-дискурсивного сходства для запроса и ответов.
3. Разработанная модель применена в задаче классификации текстовых данных. На основе предложенной модели реализован

численный метод, использующий ядерные функции. Применение модели позволяет устранить недостатки существующих моделей благодаря ранее не применявшемуся в задачах классификации абзацев использованию дискурсивной информации.

4. Разработано на базе предложенной модели таксономическое представление коллекции текстовых данных в виде решетки замкнутых структурных синтактико-дискурсивных описаний. Полученное представление применено в задаче кластеризации текстовых данных и позволяет улучшить результаты, достигаемые альтернативными моделями.
5. Разработана на основе модели текстов и теории решеток замкнутых описаний оригинальная модель тождественных денотатов для формальных описаний. Предложены численный метод и алгоритм построения связей типа «та же сущность», использующие разработанную модель. Новизна метода заключается в использовании оригинального индекса ранжирования замкнутых формальных описаний для нахождения денотатов.

Теоретическая значимость работы заключается в разработке принципиально новых моделей и методов: синтактико-дискурсивной модели текстов, позволяющей представлять текстовые абзацы в виде графов (полное описание) и лесов (приближенное описание) и вычислять сходство между текстами, таксономического представления текстовых данных, модели и метода выявления тождественных денотатов для формальных описаний.

Практическая ценность подтверждена экспериментами по оценке релевантности поиска по сложным запросам, обучению на текстовых абзацах, выявлению тождественных денотатов.

Эксперименты продемонстрировали улучшение по сравнению с существующими аналогами. Разработанные алгоритмы и методы были успешно **внедрены** в реальных проектах, а также использованы в преподавательской деятельности Департамента анализа данных и искусственного интеллекта Факультета компьютерных наук НИУ ВШЭ. Компания ООО «ФОРС-Центр разработки» применила метод классификации текстовых абзацев в проекте оценки пользовательских предпочтений. Компания Авикомп внедрила метод выявления тождественных денотатов для оптимизации прикладной онтологии. Все разработанные методы были реализованы в виде программного комплекса, предназначенного для решения исследовательских и прикладных задач.

Достоверность полученных результатов подтверждена строгостью построенных математических моделей, экспериментальной проверкой результатов численных расчетов и практической эффективностью программных реализаций.

Апробация результатов работы. Основные результаты работы обсуждались и докладывались на следующих научных конференциях и семинарах:

1. 9-й международной конференции «Интеллектуализация обработки информации» (ИОИ-2012), Будва, Черногория.
2. Семинаре по анализу формальных понятий и информационному поиску (FCAIR-2013) в рамках 35-й европейской конференции по информационному поиску (ECIR-2013), Москва, Россия.
3. 11-й международной конференции по анализу формальных понятий (ICFCA-2013), Дрезден, Германия.
4. 8-й международной конференции по компьютерной лингвистике ДИАЛОГ-2013, Москва, Россия.

5. 3-м семинаре по представлению знаний в виде графов (GKR-2013) в рамках 23-й объединенной международной конференции по искусственному интеллекту (IJCAI-2013), Пекин, Китай.
6. 7-й международной конференции по компьютерной лингвистике RANLP-2013, Хисаря, Болгария.
7. 8-й международной конференции по компьютерной лингвистике RANLP-2015, Хисаря, Болгария.
8. Ежегодном весеннем симпозиуме ассоциации искусственного интеллекта (2014 AAAI Spring Simposium).
9. 14-й международной конференции по интеллектуальной обработке текста и компьютерной лингвистике CICLING-2014, Катманду, Непал.
10. 15-й международной конференции по интеллектуальной обработке текста и компьютерной лингвистике CICLING-2015, Каир, Египет.
11. 52-й международной конференции Ассоциации компьютерной лингвистики ACL-2014, Балтимор, США.
12. 53-й международной конференции Ассоциации компьютерной лингвистики ACL-2015, Пекин, Китай.

Публикация результатов. Основные результаты работы изложены в 15 научных статьях. 12 статей опубликованы в рецензируемых трудах международных конференций, 3 статьи опубликованы в журналах из списка ВАК.

Содержание. Диссертация состоит из введения, 5 глав, заключения, списка литературы и приложений.

Во введении раскрывается актуальность темы диссертации, формулируются проблемы исследования, предмет исследования, определяется цель работы, описываются методы исследования,

излагаются основные научные результаты, обосновывается теоретическая и практическая значимость работы, даётся общая характеристика исследования.

В первой главе рассматриваются теоретические основы используемых в дальнейшем моделей и методов и описываются особенности моделирования текстовых данных. Приводятся основные определения, связанные с частично упорядоченными множествами и решетками, решетками замкнутых описаний, синтаксическими и дискурсивными моделями представления текста. Также рассматриваются некоторые подходы к структурному обучению на текстовых данных. Вводится модель структурного представления текстовых абзацев – чаща разбора.

Во второй главе описывается графовая модель текстовых абзацев, обобщающая чащу разбора, и её применение в задаче информационного поиска (для английского языка). Рассматриваются методы вычисления полного и приближенного структурного сходства текстовых абзацев, определяется проекция структурного представления текстового абзаца в виде расширенных синтаксических групп. Проводится анализ полученных результатов, демонстрируется преимущество, достигаемое за счет вычисления сходства на абзацах, производится сравнение методов, основанных на полном и приближенном сходстве. Также в главе определяется узорная структура (решетка замкнутых структурных описаний) на чащах разбора и их проекциях. Описывается применение построенной модели для иерархической кластеризации текстовых абзацев, источником которых может служить, например, поисковая выдача.

В третьей главе описывается применение построенной модели для задачи обучения с учителем на текстовых абзацах (для

английского языка), основанное на использовании ядерных функций (kernels) в методе опорных векторов (SVM). Производится сравнение с существующей моделью (Moschitti), не использующей информацию о связях между предложениями абзаца. Демонстрируется преимущество применения новой модели в нескольких прикладных задачах классификации: классификации поисковых результатов, классификации технических документов.

В четвертой главе рассматривается задача выявления тождественных денотатов для случая формальных описаний, построенных на основе предварительно обработанных текстовых данных. Предлагается модель тождественных денотатов для формальных описаний и метод, позволяющий устанавливать связи типа «та же сущность» между формальными описаниями, выделяемыми из текста. Метод основан на применении фильтрации решеток формальных понятий. Производится сравнение данного метода с альтернативными методами на нескольких наборах данных: сгенерированных и полученных из реального приложения. Демонстрируется улучшение, достигаемое за счет применения нового метода.

В пятой главе приводится описание программных комплексов, реализующих разработанные в исследовании модели и методы. Рассматриваются комплекс FCART, предназначенный для анализа данных с помощью методов анализа формальных понятий, а также программный комплекс, предназначенный для обработки чащ разбора. Описывается архитектура комплексов и применение в задачах исследования.

В приложении приводятся основные фрагменты кода программных комплексов.

1. Теоретические основы моделирования

1.1 Моделирование текстовых данных

Анализ и моделирование естественно-языковых текстовых данных – особая ветвь анализа данных, выделенная в отдельную научную область – компьютерную лингвистику. Эту область часто также называют обработкой текстов на естественном языке (Natural Language Processing). В качестве отличительных особенностей текста как объекта моделирования и анализа можно перечислить:

1. Известные априори закономерности, которым подчиняется текст.
2. Нечеткий характер наблюдаемых закономерностей, большое количество исключительных ситуаций.
3. Наличие нескольких вкладывающихся друг в друга уровней анализа и представления текста.
4. Ощутимое изменение языковой среды во времени.
5. Большие объемы доступных, но разнородных данных для анализа.
6. Доступность экспертной оценки (любой носитель языка) при верификации модельных экспериментов.

Приведенные выше особенности накладывают ряд ограничений и требований на разрабатываемые модели текстовых данных. Такого рода модели должны:

1. Учитывать реальные закономерности, наблюдаемые в текстах.
2. Учитывать формальные правила языка.
3. Быть достаточно гибкими, позволяя осуществлять настройку и доработку с учетом изменений в языковой среде.
4. Иметь привязку к определенным уровням представления текстовых данных.

Уровни моделирования текста можно расположить (в порядке возрастания уровня абстракции) следующим образом:

1. Графематический. Текст рассматривается как последовательность символов. Известно, что группы символов образуют слова или лексемы. Основная задача анализа на данном уровне – выявление лексем.
2. Морфологический. Текст представляется в виде последовательности слов и словоформ. Анализируются морфологические характеристики словоформ: леммы и грамматические свойства.
3. Синтаксический. На данном уровне рассматриваются синтаксические связи между словами в предложении или синтаксической группе.
4. Семантический.

4.1 Семантические связи внутри предложения. Анализируются семантические связи внутри предложения (семантические роли, синонимы и т.д.)

4.2 Семантические связи между предложениями (и частями сложных предложений). Анализируются так называемые дискурсивные связи: анафора, риторические отношения и т.д.

Выбор конкретного уровня моделирования текста предполагает использование (или полноценное определение в рамках новой модели) моделей для более «низких» уровней. Например, работая с предложениями, мы предполагаем, что обладаем некими моделями, позволяющими выделять отдельные слова из текстового массива, определять для этих слов части речи и т.д.

В диссертационной работе предлагается модель текста (подробнее см. раздел 2.2), относящаяся к синтаксическому (подробнее см. раздел 1.4.2) и семантическому (подробнее см. раздел 1.4.3) уровням, причем на семантическом уровне рассматриваются в первую очередь дискурсивные связи. Важно отметить, что модель включает в себя не только полное, но и приближенное (так называемая проекция, подробнее см. 2.5.1), более эффективное с вычислительной точки зрения представление текстового абзаца, а также ассоциативную и коммутативную операция вычисления сходства между текстовыми абзацами (подробнее см. раздел 2.4). Именно эти особенности модели и обусловливают её новизну по сравнению с уже существующими моделями (подробнее см. раздел 1.4.4) и позволяют применять её в прикладных задачах.

Помимо модели текстового абзаца в диссертации также предлагается новая модель тождественных денотатов (подробнее см. раздел 4), формально описывающая представление и обработку одного из типов дискурсивных связей, существующих внутри текстового абзаца.

1.2 Анализ формальных понятий и решетки замкнутых описаний

Одной из активно применяемых в исследовании математических теорий является анализ формальных понятий [26] и его расширение – решетки замкнутых описаний. Эта область сочетает в себе несколько удобных качеств, которые хорошо подходят, в частности, для работы с текстами. Во-первых, она позволяет работать с формальными описаниями произвольной степени детализации. Во-вторых, позволяет абстрагироваться от конкретного смысла и значения этих описаний, после того как сформулированы несколько простых правил работы с ними (в общем случае достаточно лишь

операции вычисления сходства, обладающей заданными свойствами). В-третьих, благодаря концепции так называемых замкнутых описаний, позволяет использовать мощный и интуитивно понятный аппарат теории решеток [1]: частичных порядков с дополнительными свойствами. Решетка одновременно является и весьма удобным моделью представления знаний, допускающим различные уровни детализации, и весьма проработанным и развитым средством для работы с данными.

Эти свойства делают решетки весьма привлекательными в плане применения к задачам обработки текста, поскольку уже существуют и известны самые разные способы и модели, позволяющие построить формальное описание текста на синтаксическом и семантическом уровне.

1.2.1 Частично упорядоченные множества и решетки

Ниже приведены основные определения из теории решеток [1], широко используемой как в теоретических, так и в прикладных областях дискретной математики, в частности, в анализе формальных понятий [26].

Определение 1.1. Бинарное отношение $\leq$ на некотором множестве S называется отношением (*нестрогого) частичного порядка*, если для $s,t,u \in S$:

1. $s \leq s$ (рефлексивность);
2. Если $s \leq t$ и $t \leq s$, то $s = t$ (антисимметричность);
3. Если $s \leq t$ и $t \leq u$, то $s \leq u$ (транзитивность).

Множество S с определённым на нем отношением частичного порядка $\leq$ (частично упорядоченное множество) обозначается $(S,\leq)$.

Если $s \leq t$, то говорят, что элемент s меньше, чем t, или равен ему. Если для s не существует t, такого что $s \leq t$, то s называют максимальным элементом S (относительно $\leq$). Если $s \leq t$ и $s \neq t$, то пишут $s < t$ и говорят, что s строго меньше, чем t.

Определение 1.2. Пусть $(S, \leq)$ частично упорядоченное множество. Элемент $l \in S$ называется *соседом снизу* элемента $u \in S$, если $l < u$ и $\neg \exists v \in S : l < v < u$. В этом случае u называется *соседом сверху* l (обозначается $l \preceq u$). Направленный граф отношения $\preceq$ называется графом покрытия.

Графически конечное частично упорядоченное множество $(S, \leq)$ может быть представлено с помощью диаграммы частичного порядка [1]. Элементы S изображаются в виде точек. Если $l \preceq u$, то u размещается «над» l (вертикальная координата u больше вертикальной координаты l), и две точки соединяются линией.

Определение 1.3. *Верхней гранью* подмножества X в упорядоченном множестве S называется элемент $l \in S$, такой что $l \geq x$ для всех $x \in X$.

Точная верхняя грань множества X (называемая также наименьшей верхней гранью или супремумом) множества X (обозначается $\sup X$) есть верхняя грань l такая, что $l \leq l_1$ для любой верхней грани l_1 подмножества X.

Двойственным образом (с заменой $\geq$ на $\leq$) определяется понятие точной (наибольшей) нижней грани или инфимума $\inf X$.

Определение 1.4. Бинарная операция $\sqcap$: $S \times S \to S$ называется *полурешёточной*, если для некоторого $e \in S$ и любых $s, t, u \in S$:

1. $s \sqcap s = s$ (идемпотентность);

2. $s \sqcap t = t \sqcap s$ (коммутативность);

3. $(s \sqcap t) \sqcap u = s \sqcap (t \sqcap u)$ (ассоциативность);

4. $s \sqcap e = e$.

Для $X = \{x_1, \ldots, x_n\} \subseteq S$ и $n \in \mathrm{N}$ мы пишем $\sqcap X$ вместо $x_1 \sqcap \ldots \sqcap x_n$.

Если $X = \{x\}$, то $\sqcap X = x$.

Определение 1.5. Множество S с определённой на нем полурешёточной операцией $\sqcap$ называется *полурешёткой* $(S, \sqcap)$.

Полурешёточная операция $\sqcap$ задает два частичных порядка $\sqsubseteq$ и $\sqsupseteq$ на S ($s, t \in S$): $s \sqsubseteq t \Leftrightarrow s \sqcap t = s$ и $s \sqsupseteq t \Leftrightarrow s \sqcap t = t$.

Тогда множество с определённой на нем полурешёточной операцией $(S,\sqcap)$ будем называть *нижней полурешёткой* (относительно частичного порядка $\sqsubseteq$) и *верхней полурешёткой* (относительно частичного порядка $\sqsupseteq$).

Определение 1.6. Пусть $(S,\sqcap)$ — полурешётка. Множество $C \subseteq S$ называется *системой замыканий* [26] или *семейством Мура* [1] (относительно $\sqcap$), если $\forall c, d \in C$: $c \sqcap d \in C$.

Очевидно, что система замыканий (относительно $\sqcap$) C с определённой на ней операцией, $\wedge$: $C \times C \to C$ и $c \wedge d = c \sqcap d$, образует полурешётку.

Определение 1.7. Упорядоченное множество $(L,\leq)$ с определёнными на нем полурешёточными операциями $\wedge$ и $\vee$ называется *решёткой*, если $(L,\wedge)$ и $(L,\vee)$ являются, соответственно, нижней и верхней полурешётками (относительно $\leq$).

Операции $\wedge$ и $\vee$ называют операциями взятия точной нижней и верхней грани в решётке или инфимума и супремума, соответственно.

Определение 1.8. *Подрешёткой* решётки L называется подмножество $X \subset L$ такое, что если $a \in X$, $b \in X$, то $a \wedge b \in X$ и $a \vee b \in X$.

Полурешёточные операции $\wedge$ и $\vee$ удовлетворяют в решётках следующему условию: $x \wedge (x \vee y) = x \vee (x \wedge y) = x$ (поглощение).

Из любой конечной полурешётки можно получить решётку добавлением одного (максимального или минимального в зависимости от типа полурешетки) элемента.

Решётка называется *полной*, если у каждого подмножества его элементов есть супремум и инфимум (всякая конечная решётка является полной).

Определение 1.9. Пусть $(S,\leq S)$ и $(T,\leq T)$ — частично упорядоченные множества. Пара отображений ϕ: $S \mapsto T$ и $\psi : T \mapsto S$ называется *соответствием Галуа* между частично упорядоченными множествами $(S,\leq S)$ и $(T,\leq T)$, если для любых $s \in S$ и $t \in T$:

1. $s_1 \leq_S s_2 \Rightarrow \phi(s_2) \leq_T \phi(s_1)$;
2. $t_1 \leq_T t_2 \Rightarrow \psi(t_2) \leq_S \psi(t_1)$;
3. $s \leq_S \psi(\phi(s))$
4. $t \leq_T \phi(\psi(s))$.

1.2.2 Анализ формальных понятий

Анализ формальных понятий (АФП) [26] - это прикладная ветвь теории решеток [1]. Основные сущности АФП были формально описаны Рудольфом Вилле в 1982 году. С точки зрения анализа данных, методы, основанные на анализе формальных понятий, относятся к методам бикластеризации (объектно-признаковой кластеризации). В АФП рассматриваются не кластеры объектов,

оторванных от исходного описания, а группы объектов и признаков, сильно связанных друг с другом.

Определение 1.10. *Формальный контекст* K есть тройка (G,M,I), где G - множество, называемое множеством *объектов*, M - множество, называемое множеством *признаков*, $I \subseteq G \times M$ - бинарное отношение.

Отношение I интерпретируется следующим образом: для $g \in G$, $m \in M$ gIm выполнено тогда и только тогда, когда объект g обладает признаком m.

Определение 1.11. Для формального контекста $K = (G,M,I)$ и произвольных $A \subseteq G, B \subseteq M$ определена пара отображений:

$$A' = \{m \in M \mid g \,\mathrm{Im}\, \forall g \in A\},\ B' = \{g \in G \mid g \,\mathrm{Im}\, \forall m \in B\}.$$

Эти отображения задают *соответствие Галуа* между частично упорядоченными множествами $(2^G, \subseteq)$ и $(2^M, \subseteq)$, а операторы $(\cdot)''$ являются *операторами замыкания* на G и M. Иными словами, для произвольного $A \subseteq G$ или $A \subseteq M$ имеют место следующие соотношения [26]:

1. $A \subseteq A''$ (экстенсивность),
2. $A''' = A'$ (идемпотентность),
3. если $A \subseteq C$, то $A'' \subseteq C''$ (изотонность).

Определение 1.12. *Формальное понятие* формального контекста $K = (G,M,I)$ есть пара (A,B), где $A \subseteq G, B \subseteq M$, $A' = B, B' = A$. Множество A называется *объёмом*, а B - *содержанием* понятия (A,B).

Для двух формальных понятий *(A,B)* и *(C,D)* некоторого контекста *(A,B)* называется *подпонятием (C,D)*, если $A \subseteq C$ (эквивалентно $D \subseteq B$). В этом случае *(C,D)* является *надпонятием (A,B)*.

Множество формальных понятий контекста *K*, упорядоченных по вложению объемов (содержаний), образует *решетку формальных понятий* $\beta(K)$.

1.2.3 Решетки замкнутых описаний

АФП преобразует формальный контекст, представленный как бинарное отношение, в решетку формальных понятий, но во многих случаях исследуемые «объекты» могут иметь более сложное описание, чем множество некоторых наперед заданных признаков. Например, *исследуя множество объектов, возможно ли их исследовать без выделения специальных бинарных признаков?* Узорные структуры (pattern structures) дают ответ на этот вопрос, являясь расширением АФП для работы со сложными данными [90], такими как данные, описываемые численными значениями, множествами последовательностей или графов.

Определение 1.13. *Узорная структура* – это тройка $(G, (D,\sqcap), \delta)$, где G – множество объектов, $(D,\sqcap)$ – полная полурешетка всевозможных описаний, а $\delta: G \to D$ – функция, которая сопоставляет каждому объекту из множества G его описание из D.

Полурешеточная операция $\sqcap$ соответствует операции сходства между двумя описаниями.

1.2.4 Проекции решеток замкнутых описаний

Поскольку размер решетки узорных понятий может быть существенным, а сама решеточная операция – асимптотически

сложной (например, для построения узорной структуры на графах необходимо вычислять изоморфизм подграфов), построение решетки узорных понятий может занимать существенное время. Для уменьшения этого времени были введены проекции узорных структур [90]. Проекция может быть рассмотрена как некоторый фильтр полурешетки описания с определенными математическими свойствами. Эти свойства позволяют доказать, что существует инъекция между понятиями спроецированной и исходной решеток.

Определение 1.14. *Проекция узорной структуры* – это функция $\psi: D \to D$, которая является монотонной $(x \sqsubseteq y \Rightarrow \psi(x) \sqsubseteq \psi(y))$, сжимающей $(\psi(x) \sqsubseteq x)$ и идемпотентной $(\psi(\psi(x)) = \psi(x))$ [90].

Узорная структура проецируется следующим образом. Мы должны спроецировать функцию – описание объектов, а также полурешетку описаний:

$\psi((G,(D,\sqcap),\delta)) = (G,(D_\psi,\sqcap_\psi),\psi \circ \delta)$, где $D_\psi = \psi(D) = \{d \in D | \exists d^* \in D: \psi(d^*) = d\}$ и $\forall x,y \in D, x \sqcap_\psi y := \psi(x \sqcap y)$.

1.3 Прикладные онтологии

Прикладные онтологии являются одной из наиболее удобных моделей для структурного представления знаний, извлеченных из текста. Они позволяют хранить данные об именованных сущностях, семантических связях и свойствах текстовых фрагментов. В нашем исследовании они применяются в качестве исходных данных задаче построения кореферентных связей в предобработанном тексте.

Введем формальное определение прикладной онтологии [27].

Определение 1.15. *Структура онтологии* - это 6-ка вида $O = (C, P, A, H^C, prop, attr)$. C - это множество всех классов онтологии, P – множество всех отношений, заданных на онтологии,

A - множество всех атрибутов, H^C - это рефлексивное антисимметричное транзитивное бинарное отношение, называемое таксономией или иерархией классов, $H^C \subseteq C \times C$, запись $(D,B) \in H^C$ означает, что класс D является подклассом B. Функция $prop : P \to C \times C$ задает множество всех отношений между классами, не относящихся к таксономии (их обычно называют горизонтальными). Функция $attr : A \to C$ определяет, какими атрибутами наделен каждый класс.

Определение 1.16. *Экземпляр онтологии* (или просто онтология) - это 6-ка вида $MD = (O, I, L, inst, instr, instl)$. O - это структура онтологии, I - множество всех идентификаторов, используемых в данном экземпляре (множества I, P, C не пересекаются), L - множество значений атрибутов. Функция $inst : C \to 2^I$ задает множество экземпляров классов, называемых объектами онтологии. Функция $instr : P \to 2^{I \times I}$ задает множество экземпляров отношений. Функция $instl : A \to 2^{I \times L}$ задает значения атрибутов для каждого объекта.

В простейшем случае можно считать, что все атрибуты – это литералы. В более сложном случае каждый атрибут принадлежит к определенному домену, а вместо одного множества L рассматривается набор используемых доменов.

1.4 Модели представления текста

1.4.1 Мешок слов

Модель «мешка слов»[18, 69] («bag-of-words») дает упрощенное представление текста, применяемое, в частности, в задаче информационного поиска. В этой модели текст представляется как неупорядоченный набор слов (или словосочетаний) без учета

грамматики и порядка слов. «Мешок слов» часто используется для классификации документов, где частота встречаемости слова используется в качестве признака для обучения классификатора. Основным изменяемым параметром является используемый вес – наиболее распространена характеристика *tf-idf* – частота встречаемости слов и обратная частота документа.

TF – отношение числа вхождения некоторого слова к общему количеству слов в документе: $tf(t,d) = \frac{n_i}{\sum_k n_k}$, *IDF* – величина, обратная к частоте, с которой слово встречается в документах, с помощью *IDF* снижается вес общеупотребительных слов. $idf(t,D) = \log \frac{|D|}{|(d_i \supset t_i)|}$, где D – коллекция документов, d – конкретный документ, t – конкретное слово. Мера TF-IDF есть произведение: $tfidf(t,d,D) = tf(t,d) * idf(t,D)$.

Большое значение этой меры соответствует словам, часто встречающимся в одном документе и редко – в других документах коллекции.

1.4.2 Деревья синтаксического разбора

Дерево синтаксического разбора (syntactic parse tree) – это упорядоченное дерево, которое отражает синтаксическую структуру предложения или строки согласно некоторой формальной грамматике. Выделяют два основных класса: деревья составляющих (constituency tree) и деревья зависимостей (dependency tree). Важно отличать деревья синтаксического разбора от абстрактных синтаксических деревьев – первые точнее отражают синтаксис входного языка, в то время как вторые скорее применимы к полностью формализованным областям, таким как языки программирования. Деревья

синтаксического разбора используются и для компьютерных языков, и для обработки текстов на естественных языках. Важной характеристикой деревьев, с точки зрения нашего исследования, является наличие хорошо развитых методов и программных средств, предназначенных для их автоматического построения. В первую очередь, это касается английского языка, хотя для русского также существуют методы построения деревьев, использующие деревья зависимостей. В экспериментах используются деревья составляющих для английского языка, их построение осуществляется с помощью методов, разработанных группой из Стэнфорда [23].

1.4.2.1 Деревья составляющих

Деревья составляющих отвечают фразовой структуре предложения (синтаксическим группам). Подход на основе составляющих использует идею контекстно-свободной грамматики. Основные классы таких грамматик были разработаны Ноамом Хомским [57]. В деревьях составляющих внутренние вершины помечаются нетерминальными символами, отвечающими грамматическим категориям, в листьях же располагаются непосредственно слова. Деревья составляющих применяются в первую очередь для языков со строгим порядком слов в предложении.

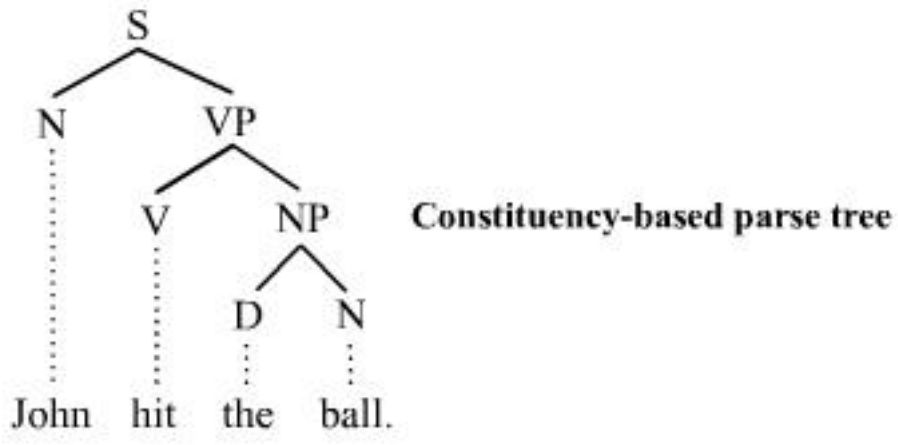

Рис. 1.1. Дерево составляющих для предложения «John hit the ball» [17, 57]

Классический набор нетерминальных символов или тегов для английского языка, представлен, например, в корпусе Penn Treebank [56].

Таблица 1.1. Основные нетерминальные символы, используемые в Penn Treebank

Тэг	**Описание**	**Пример**
CC	coordinating conjunction	And
CD	cardinal number	1, third
DT	Determiner	The
EX	existential there	*there* is
FW	foreign word	d'hoevre
IN	preposition/subordinating conjunction	in, of, like
JJ	Adjective	Green
JJR	adjective, comparative	Greener
JJS	adjective, superlative	Greenest
LS	list marker	1)
MD	Modal	could, will
NN	noun, singular or mass	Table
NNS	noun plural	Tables
NNP	proper noun, singular	John
NNPS	proper noun, plural	Vikings
PDT	predeterminer	Both, the
POS	possessive ending	friend*'s*
PRP	personal pronoun	I, he, it
PRP$	possessive pronoun	my, his
RB	Adverb	however, usually, naturally, here, good
RBR	adverb, comparative	Better
RBS	adverb, superlative	Best
RP	Particle	give *up*
TO	To	*to* go, *to* him
UH	Interjection	uhhuhhuhh
VB	verb, base form	Take
VBD	verb, past tense	Took
VBG	verb, gerund/present participle	Taking

VBN	verb, past participle	Taken
VBP	verb, sing. present, non-3d	Take
VBZ	verb, 3rd person sing. present	Takes
WDT	wh-determiner	Which
WP	wh-pronoun	who, what
WP$	possessive wh-pronoun	Whose
WRB	wh-abverb	where, when

1.4.2.2 Деревья зависимостей

Деревья зависимостей базируются на грамматике зависимостей – классе современных синтаксических теорий, которые основаны на отношении зависимости, которое можно проследить вплоть до работ Люсьена Теньера [9]. Центром высказывания считается глагол. Все остальные синтаксические единицы зависят от глагола, возможно, не напрямую. Отличие от грамматик составляющих заключается в отсутствии вершин для фраз. Деревья зависимостей хорошо подходят для языков со свободным порядком слов, например, для русского и турецкого языка.

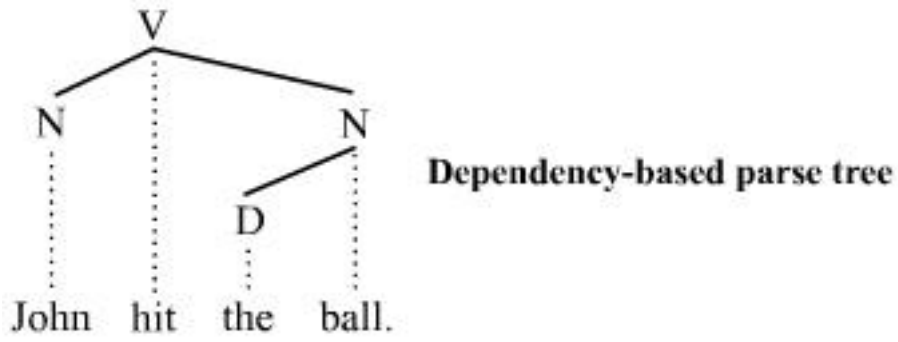

Рис. 1.2. Дерево зависимостей для предложения «John hit the ball» [17, 57]

1.4.3 Представление дискурсивных отношений между предложениями текста

1.4.3.1 Дискурсивные теории и их применение в прикладных задачах

Рассмотренные выше деревья синтаксического разбора позволяют создать структурное представление предложения на базе связей внутри этого предложения. Однако если мы рассматриваем более объемные тексты, например, абзацы, состоящие из нескольких предложений, то использования синтаксической информации оказывается недостаточно. В этом случае источником структурных связей могут служить дискурсивные теории, учитывающие смысловые отношения между фрагментами текста. Ниже описаны несколько основных наиболее востребованных на данный момент теорий, использующих различные источники данных и отображающих различные типы отношений. Несмотря на значительное количество разработанных (для английского языка) теорий, реально применимыми на практике оказываются лишь немногие из них. В исследовании использовались теория риторических структур и теория речевых актов (в части коммуникативных действий). Их выбор был обусловлен относительной простотой этих теорий, а также наличием алгоритмического и математического описания, позволяющего создать полностью автоматическое построение описываемых в этих теориях связей без привлечения экспертов как на этапе формирования описываемой в исследовании модели текстов, так и на этапе применения модели в прикладных задачах. Тем не менее, необходимо отметить, что модель допускает добавление связей других типов, ограничением является лишь необходимость установления каждой

связи между двумя вершинами деревьев разбора для сохранения графовой структуры представления текста.

Важно отметить, что дискурсивные связи уже продемонстрировали свою полезность в ряде прикладных задач, связанных с анализом текстовых данных [20]. В частности, они использовались для повышения качества машинного перевода [84, 85, 86], проведения анализа связности текста [87], повышения точности поиска для семантически связанных вопросов [88] и для вопросов специального типа [89]. Дискурсивные связи применялись и в нескольких исследованиях, связанных с русским языком, однако рассматривалась только устная речь [21].

1.4.3.2 Теория риторических структур

Теория риторических структур [74, 75] дает общее представление об организации текста на естественном языке (в исследовании рассматривается реализация для английского языка). Этот метод полезен с лингвистической точки зрения тем, что позволяет описывать структуру текста в терминах отношений между его частями.

Выделяют несколько основных преимуществ данной теории: она нечувствительна к изменению размера текста, обеспечивает всесторонний анализ текста, описывает отношения функционально. Теория риторических структур позволяет описывать зависимости между предложениями в тексте вне зависимости от грамматических и лексических признаков.

При описании теории риторических структур используют 4 основных сущности:

1. Отношения;

2. Схемы;
3. Реализации схем;
4. Структуры.

Отношения – конкретные бинарные связи, которые устанавливаются между частями текста. Каждое отношения связывает два непересекающихся фрагмента текста, называемых ядром (nucleus) и спутником (satellite), обозначаемых N и S соответственно. Стоит также отметить, что определение отношений основывается не на морфологических и синтаксических признаках, а на функциональных и семантических. Так, для определения отношения *Condition* (условие) недостаточно лишь только наличия слова ‘if’.

Отношение состоит из 4 компонентов:

1. Ограничения на ядро.
2. Ограничения на спутник.
3. Ограничения на сочетание ядра и спутника.
4. «Эффект».

Схемы определяют общую структуру текста. Это абстрактные образцы, состоящие из небольшого числа фрагментов текста, отношений между ними и взаимосвязей между ядрами и всеми фрагмента. Они немного похожи на грамматические правила.

В теории риторических структур выделяют 5 основных видов схем:

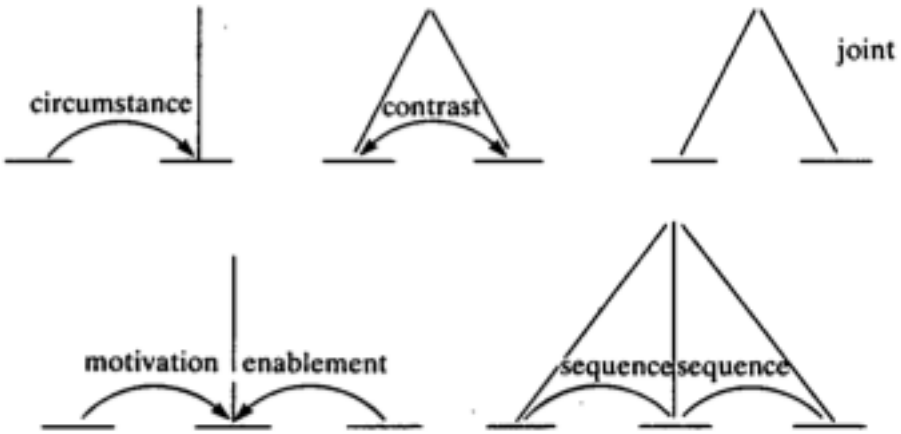

Рис. 1.3. Основные виды схем в теории риторических отношений [74]

Прямые линии отвечают атомарным фрагментам, а кривые – имеющимся отношениям (стрелка направлена от ядра к спутнику). Остальные схемы устроены так же, как схема обстоятельств (circumstance), и имеют выделенное ядро и спутник.

Многоядерные схемы используются для представления текстов, которые не сводятся к одноядерной модели. В отношении *contrast* всегда есть ровно два ядра, а в отношениях *sequence* и *joint* их может быть неограниченное число (для *sequence* они все последовательно связаны). При этом данные фрагменты объявляются ядрами по договоренности, несмотря на отсутствие соответствующих спутников.

Схемы в реальных текстах могут быть составлены не полностью, допустимы некоторые отклонения от определения. Существует 3 основных вида *реализаций схем*:

1. Неупорядоченные фрагменты: в схеме нет ограничений на порядок ядро-спутник

2. Необязательные отношения: для схемы со многими отношениями – все одинарные необязательны, но хотя бы одно должно выполняться
3. Повторяющиеся отношения: отношение, которое есть часть схемы, может быть повторено сколько угодно раз при применении этой схемы.

Результат *структурного анализа* – такой набор реализации схем, что выполняются следующие ограничения:

1. Полнота: есть одна реализация схемы, чьи фрагменты покрывают весь текст.
2. Связность: любой фрагмент текста либо является минимальной единицей, либо является частью другой реализации схемы.
3. Уникальность: каждая реализации схемы состоит из разных фрагментов текста, а для многоядерных схем каждое отношение применяется к разным фрагментам.
4. Целостность: все вместе фрагменты каждой реализации схемы образуют один текст.

Стоит отметить, что ограничения 1-3 обеспечивают, что результатом анализа станут не произвольные структуры, а деревья.

Ниже приведены примеры основных отношений:

- Circumstance (обстоятельство)
- Solutionhood (принятие решения)
- Elaboration (уточнение)
- Background (предпосылка)
- Enablement (снятие запрета)
- Motivation (побуждение)

- Evidence (свидетельство)
- Justify (подтверждение)
- Volitional Cause (волевая причина)
- Non-Volitional Cause (неволевая причина)
- Volitional Result (волевой результат)
- Non-Volitional Result (неволевой результат)
- Purpose (намерение)
- Antithesis (антитеза)
- Concession (уступка)
- Condition (условие)
- Otherwise (иначе)
- Interpretation (толкование)
- Evaluation (оценка)
- Restatement (подтверждение)
- Summary (резюме)
- Sequence (очередность)
- Contrast (контраст)

Рассмотрим в качестве примера отношение *evidence*.

Ограничения на ядро: читатель верит в ядро недостаточно с точки зрения автора. *Ограничения на спутник*: читатель верит в его утверждение. *Ограничения на сочетание ядро-спутник*: прочтение спутника увеличивает доверие читателя к ядру.

Эффект: читатель больше доверяет ядру. *Положение эффекта*: ядро.

В качестве конкретного употребления отношения можно рассмотреть следующий отрывок текста:

‘*The program as published for calendar year 1980 really works. In only a few minutes, I entered all the figures from my 1980 tax return and got a result which agreed with my hand calculations to the penny.’*

Первое предложение – ядро, второе и третье образуют спутник. Последние два предложения подтверждают высказывание в первом.

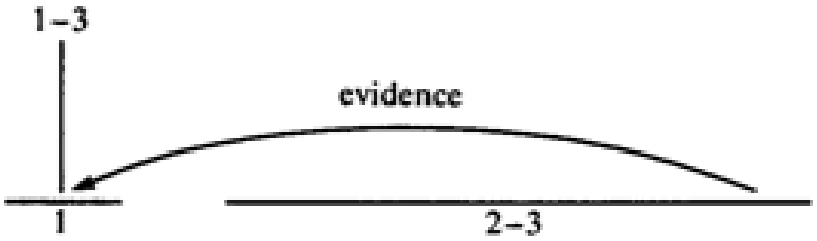

Рис. 1.4. Пример реализации риторического отношения evidence [74]

1.4.3.3 Теория речевых актов

Теория речевых актов была разработана Джон Остином и его учеником Джоном Серлем для английского языка [82].

Основной функцией языка считается формулирование истинных или ложных высказываний, например, «снег белый». Однако не все предложения укладываются в такую схему («Доброе утро»). В связи с этим вводится понятие перформативных высказываний – реализации действия. Они не совпадают с описаниями, для них невозможно ввести истинность. Эти высказывания отвечают неким действиям, например, приказу или предупреждению. Для перформативных высказываний вводится оценка выполнимости/невыполнимости. Перформативы обычно вводятся с помощью соответствующих глаголов, обычно у них есть субъект в единственном числе первом лице и стоят в простом настоящем времени, активном залоге и изъяснительном наклонении. Такие высказывания скорее указывают

на действие («Объявляю вас мужем и женой»). Перформативы бывают явные (с соответствующим глаголом, «Я обещаю прийти на вечеринку») и неявные (без глагола, «Я приду на вечеринку»).

Выделяют три типа речевых актов:

1. Локутивный – констатация. Пример: «Он сказал ей: возьми куртку»
2. Иллокутативный – выражает намерение. Пример: «Он советовал ей взять куртку»
3. Перлокутивный – воздействие на поведение другого человека. Пример: «Он уговорил ее взять куртку»

На практике сложно отнести акт к какому-то одному типу, то есть в чистом виде все типы не встречаются.

Для выделения классов используют иллокутивную цель, направление отношений между содержанием акта и текущей обстановкой и другие факторы.

Существуют следующие типы иллокутивных целей:

- информативная («Поезд пришел»)
- побуждение («Пропустите!»)
- принятие обязательств («Обещаю не опаздывать»)
- эмоциональное состояние («Прошу прощения»)

1.4.3.4 Семантическая организация данных

Теория семантической организации данных (Database Semantics)[127] описывает семантические отношения, возникающие при взаимодействии людей посредством языка. Одной из частей теории является функциональная декларативная модель «разговаривающего робота», позволяющая описать процессы

мышления, слуха (восприятия информации) и говорения, свойственные людям. Теория включает в себя модель языка (используется специально разработанная модель Slim) и модель грамматики. Важной частью теории является представление, используемое для хранения высказываний – так называемые проплеты (proplets) или центральные части высказываний, описываемые с помощью упорядоченных наборов атрибутов.

1.4.3.5 Теория представления дискурса

Теория представления дискурса (Discourse representation theory) [128] является попыткой описания дискрусивных отношений и дискурсивной структуры текста в терминах формальной логики. Авторы вводят специальное абстрактное логическое представление: *структуры для представления дискурса* (DRS). Эти структуры используются для формального описания *интерпретации* текста. Дискурсивные структуры строятся на основе применения к тексту набора предварительно заданных *правил построения дискурсивных отношений*. Эти правила апеллируют к синтаксической структуре предложений и к деревьям синтаксического разбора, используя их как базис для построения отношений, отражающих смысл текста и учитывающих различные правила языка.

1.4.4 Чаща разбора

Выше были рассмотрены несколько дискурсивных теорий, позволяющих установить связи внутри текста, состоящего из нескольких предложений. Используя эти теории, можно обобщить понятие дерева синтаксического разбора на случай текстового абзаца. В работе [61] был введен термин **чаща разбора** (Parse Thicket). Чаща разбора описывает синтактико-дискурсивную структуру абзаца.

Определение 1.16. *Чащей разбора* текстового абзаца называется совокупность множества деревьев разбора для предложений абзаца и связей нескольких типов, устанавливаемых между вершинами этих деревьев. Каждая связь – это упорядоченная пара вершин деревьев разбора, причем связаны между собой могут быть как вершины одного и того же, так и разных деревьев.

Со структурной точки зрения, чаща представляет собой ориентированный граф, который включает в себя деревья разбора, а также дуги, соответствующие несинтаксическим связям.

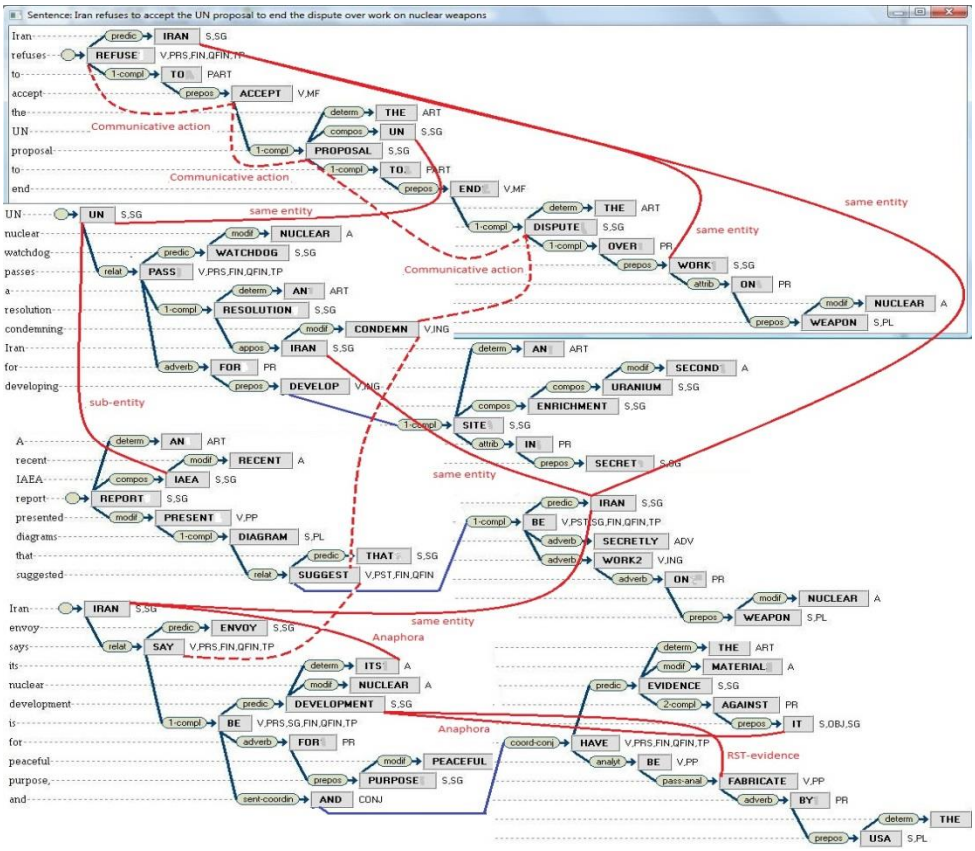

Рисунок 1.3. Пример графического представления чащи разбора

1.4.5 Теория «Смысл ⇔ Текст»

Чаща разбора представляет собой непосредственное развитие идеи синтаксических деревьев разбора на случай абзаца. При этом фактически чаща предполагает частично независимое построение

синтаксических и дискурсивных связей. Более того, разные типы дискурсивных связей также строятся независимо друг от друга. Кроме того, её применение предполагает наличие развитых инструментов, позволяющих автоматически конструировать деревья и связи между ними. Это требование в полной мере выполняется для английского языка, который и рассматривался в экспериментах. Возможной альтернативой чаще может служить представление, основанное на использовании теории «Смысл ⇔ Текст», разработанной И. Мельчуком [10]. Теория «Смысл ⇔ Текст» представляет собой описание естественного языка, понимаемого как устройство («система правил»), обеспечивающее человеку переход от смысла к тексту («говорение», или построение текста) и от текста к смыслу («понимание», или интерпретация текста). Теория постулирует многоуровневую модель языка, то есть такую, в которой построение текста на основе заданного смысла происходит не непосредственно, а с помощью серии переходов от одного уровня представления к другому. Помимо двух «крайних» уровней — фонологического (уровня текста) и семантического (уровня смысла), выделяются поверхностно-морфологический, глубинно-морфологический, поверхностно-синтаксический и глубинно-синтаксический уровни. Каждый уровень характеризуется набором собственных единиц и правил представления, а также набором правил перехода от данного уровня представления к соседним. Семантическое представление является неупорядоченным графом («сетью»), синтаксические представления являются графическим деревом («деревом зависимостей»).

Данная теория обладает целым рядом важных свойств:

1. Применима к языкам со свободным порядком слов, в том числе и к русскому языку.
2. Описывает многие отношения и связи, известные из других теорий, такие как синтаксические связи, анафора, семантико-коммуникативные связи [97] и т.д.
3. Комбинирует синтаксический и семантический уровни анализа.
4. Так же, как и чаща разбора, позволяет получить графовое представление текста (на семантическом уровне).

Очевидным недостатком такого представления является отсутствие полноценного математического описания данной теории и программной реализации (выходящей за рамки построения деревьев зависимостей). Также стоит отметить, что собственно дискурсивные связи (за исключением анафоры) в этой теории не рассматриваются. Однако данное представление является весьма перспективным с точки зрения применения модели и методов сходства, описанных в нашей работе, для других языков, помимо английского. Тем не менее, в экспериментальных исследованиях предпочтение было отдано чаще разбора.

1.5 Ядра в задаче машинного обучения

В нашей работе мы будем исследовать применение ядер на деревьях в задаче классификации коротких текстов. Предлагаемый метод допускает использование различных видов ядерных функций на деревьях: неглубоких ядер, частичных ядер и т.д. В наших экспериментах мы применяли ядра простейшего вида. Такой выбор был обусловлен рядом факторов. Во-первых, требовалась связность порождаемых подструктур, отсутствовали корректирующие коэффициенты («штрафы» за размер подструктур). Во-вторых,

постановка задачи не предполагала использование предикатных семантических связей, для исследования которых, как правило, применяются неглубокие (shallow) ядра. В-третьих, важной компонентой экспериментов было сравнение с существующим подходом к классификации предложений, использующим именно ядра с запретом на несвязность подструктур.

1.5.1 Применение ядерных функций в задачах машинного обучения

Определение 1.17. Отображение $K: X \times X \to \Re$ называется *ядром* или ядерной функцией [96], если оно обладает следующими свойствами:

1. Симметричность: $\forall x, y \in X \; K(x,y) = K(y,x)$.
2. Неотрицательная определенность: $\forall N \; \forall x_1, \ldots, x_N \in X$ матрица K, задаваемая как $K_{ij} = K\left(x_i, x_j\right)$, является неотрицательно определенной.

Ядерные функции применяются в сочетании с широким классом алгоритмов обучения, основанных на скалярном произведении в векторных пространствах. Их применение обусловлено так называемым *трюком с ядрами* (*kernel trick*) [76]. Пусть у нас имеется отображение $\phi: X \to V$, где V – пространство со скалярным произведением, например, $\Re^n$. Тогда ядро можно определить как скалярное произведение в этом пространстве: $K\left(x,y\right) = \left\langle \phi\left(x\right) \cdot \phi\left(y\right) \right\rangle_V$. Этот прием в ряде случаев позволяет заменить трудоемкие вычисления исходных отображений на элементах исходного множества на подсчет скалярного произведения. Данный прием, в частности, применяется в Методе Опорных Векторов (Support Vector Machine) [81]. Основная идея этого метода – нахождение разделяющей гиперплоскости вида $H(\bar{\bar{x}}) = \bar{\bar{w}} \cdot \bar{\bar{x}} + b = 0$, где $\bar{\bar{x}}$ -

вектор признаков, отвечающий классифицируемому объекту, а $\bar{w} \in R^n, b \in R^n$ – параметры алгоритма, обучаемые согласно принципу минимизации риска. Применение ядер позволяет использовать данный метод для объектов, имеющих сложную структуру и очень большое число свойств, не прибегая к явному выделению этих признаков. Пусть задано отображение (соответствующее выделение обучающих признаков для исходных объектов) $\phi: X \to \Re^n$. Перепишем выражение для разделяющей гиперплоскости следующим образом:

$$H(\bar{z}) = \left(\sum_{i=1}^{l} y_i \alpha_i \bar{z}_i \right) \cdot \bar{z} + b = \sum_{i=1}^{l} y_i \alpha_i \bar{z}_i \cdot \bar{z} + b = \sum_{i=1}^{l} y_i \alpha_i \phi(x_i) \cdot \phi(x) + b$$

где $y_i = \pm 1$ в зависимости от класса, $\alpha_i \in R$, $\alpha_i \geq 0$; $z_i \; \forall i \in \{1, \ldots l\}$ – примеры из обучающей выборки.

Как уже отмечалось выше, вместо применения функции ϕ можно напрямую считать $K(x_i, x)$. Ядерная функция позволяет неявно оперировать в пространствах с огромным числом признаков (потенциально бесконечномерных).

1.5.2 Некоторые виды ядер

Рассмотрим некоторые виды ядер, задаваемые для структур, представляющих текстовые строки и предложения.

1.5.2.1 Ядра для строк

Наиболее простым примером представляющей текстовую информацию структуры, для которой можно определить функцию ядра, являются строки. Строковые ядра [110] подсчитывают количество общих для двух последовательностей подстрок, возможно, с пропусками, например, могут быть опущены некоторые символы первой строки. Пропуски учитываются в весе целевых подстрок.

Пусть Σ – конечный алфавит, $\Sigma^* = \cup_{n=0}^{\infty} \Sigma^n$ – множество всех строк. Для каждой строки $\sigma \in \Sigma^*, |\sigma|$ обозначает длину строки, саму строку можно записать в виде: $s_1..s_{|\sigma|}$, где $s_i \in \Sigma$, $\sigma[i:j]$ обозначает выбор подстроки с *i*-го по *j*-й символ: $s_i s_{i+1}..s_{j-1}s_j$.

Определение 1.18. *u – подпоследовательность* в σ, если существует последовательность индексов $\bar{I} = (i_1, \ldots, i_{|u|})$,, где $1 \le i_1 < \ldots i_{|u|} \le \sigma$, такая что $u = s_{i_1}..s_{i_{|u|}}$ или, для краткости записи, $u = \sigma[\bar{I}]$.

Через $d(\bar{I})$ обозначают расстояние между первым и последним индексом подпоследовательности в оригинальной строке: $d(\bar{I}) = i_{|u|} - i_1 + 1$. С помощью $\sigma_1 \sigma_2$ обозначается конкатенация строк $\sigma_1, \sigma_2 \in \Sigma^*$.

Множество всех подстрок данного корпуса образует пространство признаков, обозначаемое $\mathcal{F} \subset \Sigma^*$. Чтобы отобразить строку σ в пространство $\mathbb{R}^\infty$, можно использовать следующий класс функций: $\phi_u(\sigma) = \sum_{\bar{I}:u=s[\bar{I}]} \lambda^{d(\bar{I})}$ для некоторого $\lambda \le 1$. Такие функции подсчитывают число вхождений u в строку σ и назначают им вес $\lambda^{d(\bar{I})}$, пропорциональный их длине. Таким образом, скалярное произведение в пространстве признаков для двух строк σ_1 и σ_2 возвращает взвешенную по частоте встречаемости и длинам сумму всех общих подпоследовательностей. Использующая введенное отображение функция ядра будет выглядеть следующим образом:

- $$SK(\sigma_1, \sigma_2) = \sum_{u \in \Sigma^*} \phi_u(\sigma_1) \cdot \phi_u(\sigma_1) = \sum_{u \in \Sigma^*} \sum_{I_1: u=\sigma_1[I_1]} \lambda^{d(I_1)} \sum_{I_2: u=\sigma_2[I_2]} \lambda^{d(I_2)} =$$
$$= \sum_{u \in \Sigma^*} \sum_{I_1: u=\sigma_1[I_1]} \sum_{I_2: u=\sigma_2[I_2]} \lambda^{d(I_1)+d(I_2)}$$

Данное ядро имеет следующие свойства:

1. Длинные подпоследовательности получают меньший вес;
2. Допускается пропуск символов исходной строки;
3. Пропуски учитываются в весовой функции $d(.)$;
4. Символами можно считать слова, тогда мы получим ядро для последовательностей слов.

1.5.2.2 Ядро на синтаксических деревьях

Основная идея ядер для деревьев [76], как и в случае со строками, заключается в том, чтобы подсчитывать количество общих подструктур для двух деревьев T_1 и T_2 без явного учета пространства всех подструктур.

Пусть $\mathcal{F} = \{f_1, f_2, \dots, f_{|\mathcal{F}|}\}$ – множество всех поддеревьев, а $\chi_i(n)$ – индикаторная функция, которая равна 1, если целевое поддерево имеет корнем вершину n. Ядро для T_1 и T_2 определяется как $TK(T_1, T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2)$, где N_{T_1} и N_{T_2} – множества вершин деревьев T_1 и T_2 соответственно, а $\Delta(n_1, n_2) = \sum_{i=1}^{|F|} \chi_i(n_1)\chi_i(n_2)$.

Функция Δ определяет количество общих фрагментов, начинающихся в вершинах n_1 и n_2, и зависит от типа фрагмента.

В простейшем случае в качестве подструктур рассматриваются множества ребер и вершин из исходного дерева, которое также образуют деревья с дополнительным ограничением: для любой вершины должны учитываться или все, или ни одна из ее дочерних вершин.

Для подсчета общего количества таких подструктур, начинающихся в вершинах n_1 и n_2, используется следующий вид функции Δ:

1. если продукции (вершина и её прямые потомки) в вершинах n_1 и n_2 не совпадают, то $\Delta(n_1, n_2) = 0$.
2. если продукции в вершинах одинаковы, и все дочерние вершины являются листьями, то $\Delta(\mathrm{n}_1, \mathrm{n}_2) = \lambda$. λ – дополнительный коэффициент, позволяющий нормализовать вычисление ядер для больших структур.
3. если продукции в вершинах совпадают и вершины не являются предтерминальными (то есть имеющими только листья в качестве дочерних вершин), то вводится рекурсивное выражение: $\Delta(\mathrm{n}_1, \mathrm{n}_2) = \lambda \prod_{\mathrm{j}=1}^{\mathrm{l}(\mathrm{n}_1)} \left(1 + \Delta\left(\mathrm{c}_{\mathrm{n}_1}(\mathrm{j}), \mathrm{c}_{\mathrm{n}_2}(\mathrm{j})\right)\right)$, где $\mathrm{l}(\mathrm{n}_1)$ – количество дочерних вершин для n_1, $c_n(j)$– j-й ребенок вершины n.

1.5.2.3 Неглубокое семантическое ядро

Данный вид ядерной функции применяется для обучения на деревьях, представляющих семантическую структуру текста [79]. Подструктуры в этом случае почти полностью совпадают со случаем ядер на синтаксических деревьях, единственное различие – вклад специального типа вершин, помеченных *null*, должен быть нулевым. Это необходимо, поскольку ядро применяется для деревьев специального вида, содержащих «слотовые» вершины, потомки которых могут быть помечены как *null*.

Алгоритм вычисления функции Δ меняется следующим образом:

1. если n_1 (или n_2) – предтерминальная вершина, и метка ее потомка – *null*, то $\Delta(\mathrm{n}_1, \mathrm{n}_2) = 0$.
2. если продукции в вершинах n_1 и n_2 не совпадают, то $\Delta(\mathrm{n}_1, \mathrm{n}_2) = 0$
3. если продукции в вершинах одинаковы, и дочерние вершины только терминальные, то $\Delta(n_1, n_2) = \lambda$.
4. если продукции в вершинах совпадают, и вершины не предтерминальные, то вводится рекурсивное выражение: $\Delta(\mathrm{n}_1, \mathrm{n}_2) = \prod_{\mathrm{j}=1}^{\mathrm{l}(\mathrm{n}_1)}(1 + \Delta\left(\mathrm{c}_{\mathrm{n}_1}(\mathrm{j}), \mathrm{c}_{\mathrm{n}_2}(\mathrm{j})\right)$ -1.

1.5.2.4 Ядро частичных поддеревьев

Если ослабить требование касательно продукций, то получатся подструктуры более общего вида, а функция Δ упростится [113]. Для двух вершин n_1 и n_2 ядро на синтаксических деревьях применяется для всех возможных подпоследовательностей потомков этих вершин.

Алгоритм подсчета функции Δ выглядит следующим образом:

1. Если метки вершин n_1 и n_2 разные, то $\Delta(n_1, n_2) = 0$.

2. В противном случае

$\Delta(n_1, n_2) = 1 + \sum_{I_1, I_2, l(I_1)=l(I_2)} \prod_{j=1}^{l(I_1)} \Delta\left(c_{n_1}\left(I_{1_j}\right), c_{n_2}\left(I_{2_j}\right)\right), \Delta(\mathrm{n}_1, \mathrm{n}_2) = 1 +$

$\sum_{\bar{\bar{\mathrm{I}}}_1, \bar{\bar{\mathrm{I}}}_2, \mathrm{l}(\bar{\bar{\mathrm{I}}}_1)=\mathrm{l}(\bar{\bar{\mathrm{I}}}_2)} \prod_{\mathrm{j}=1}^{\mathrm{l}(\bar{\bar{\mathrm{I}}}_1)} \Delta(\mathrm{c}_{\mathrm{n}_1}(\bar{\bar{\mathrm{I}}}_{1\mathrm{j}}), \mathrm{c}_{\mathrm{n}_2}(\bar{\bar{\mathrm{I}}}_{2\mathrm{j}}))$, где $\bar{\bar{\mathrm{I}}}_1 =< \mathrm{h}_1, \mathrm{h}_2, \mathrm{h}_3, \ldots >$ и $I_2 =< k_1, k_2, k_3, \ldots >$ – последовательности индексов, отвечающие упорядоченной последовательности потомков $\mathrm{c}_{\mathrm{n}_1}$ для n_1 и $\mathrm{c}_{\mathrm{n}_1}$ для n_2 соответственно, I_{1_j} и I_{2_j} указывают на j-ого потомка в соответствующей последовательности, а $l(.)$ – длина последовательности, то есть количество дочерних вершин.

Далее вводятся штрафы для глубины деревьев μ и для длины последовательности потомков λ. Итоговое выражение:

$$\Delta(n_1, n_2) = \mu\left(\lambda^2 + \sum_{I_1, I_2, l(I_1)=l(I_2)} \lambda^{d(I_1)+d(I_2)} \prod_{j=1}^{l(I_1)} \Delta\left(c_{n_1}\left(I_{1_j}\right), c_{n_2}\left(I_{2_j}\right)\right)\right),$$

где $d(I_1) = I_{1_{l(I_1)}} - I_{1_1}$ и $d(I_2) = I_{2_{l(I_2)}} - I_{2_1}$.

Таким образом, штраф накладывается на большие деревья и последовательности дочерних вершин, в которых есть пропуски.

2. Модели и методы поиска ответов на сложные запросы

2.1 Введение

Современные поисковые машины недостаточно хорошо обрабатывают запросы, состоящие из нескольких предложений. Они находят либо очень похожие документы (если таковые имеются), либо документы, сильно отличающиеся от ожидаемого результата, что делает результаты поиска не слишком полезными для пользователя. Это обусловлено тем, что для запросов, состоящих из нескольких предложений, довольно трудно построить ранжирование, основанное на данных о пользовательских «кликах» по результатам, так как число такого рода запросов практически не ограничено. Поэтому необходима лингвистическая технология, которая бы переупорядочивала потенциальные ответы, используя структурное сходство между вопросом и ответом. В нашем исследовании предлагается представление текстового абзаца, позволяющее отслеживать упомянутые структурные различия, используя не только информацию, содержащуюся в деревьях разбора, но и дискурсивную информацию, характеризующую абзац как лингвистическую структуру. Исследование ориентировано на обработку текстов на английском языке.

Использование абзацев текста в качестве запросов применяется, например, в основанных на поиске рекомендательных системах [53–55]. Рекомендательные агенты отслеживают действия пользователей чатов, блогов и форумов, комментарии пользователей на торговых сайтах и предлагают веб-документы и их фрагменты, относящиеся к решениям о покупке товара. Для формирования рекомендации агенты должны взять части текста, построить запрос для поисковой системы, запустить его с помощью API поисковой системы, такой как Yahoo

или Bing, и отфильтровать нерелевантные по отношению к решению о покупке результаты поиска. Последний шаг имеет решающее значение для разумного функционирования агента, поскольку низкая релевантность приведет к утрате доверия по отношению к механизму рекомендаций. Поэтому нахождение точной оценки сходства между двумя частями текста имеет решающее значение для успешного использования рекомендательных агентов.

Деревья синтаксического разбора являются стандартной формой представления синтаксической структуры предложений [58–60]. В нашем исследовании для представления лингвистической структуры абзаца текста используются деревья разбора, конструируемые для каждого предложения абзаца, а также обобщенная модель чащи разбора (Parse Thicket), используемая для представления абзаца.

2.2 Обобщенная модель текстового абзаца

В работе [61] отмечается теоретическая возможность построения структурного представления абзаца текста и вводится понятие чащи разбора (Parse Thicket), которая определяется как ориентированный граф, включающий в себя деревья синтаксического разбора, а также (опционально) дуги, соответствующие несинтаксическим связям. В нашем исследовании эта модель реализуется на практике, а также модифицируется и расширяется за счет добавления в неё операции обобщения абзацев текста и конкретных типов несинтаксических (дискурсивных) связей. Эта модификация позволяет применять данную модель в задачах поиска, классификации, кластеризации текстов.

Определение 2.1. Обобщенной моделью текстового абзаца P на основе «чащи разбора» называется пара $(G, \sqcap)$, где G – множество, состоящее из ориентированного графа с метками на вершинах и

ребрах («чаща разбора»), а ⊓ – операция структурного обобщения, определяемая на произвольном конечном множестве графов с метками вершин и ребер.

Определим вначале операцию обобщения для двух чащ разбора и покажем, как применение этой операции позволяет решать задачу вычисления сходства текстов и повышения релевантности поиска. Использование обобщения для оценки сходства продолжает линию структурного подхода к машинному обучению [62–65], альтернативой которому является измерение статистического сходства как расстояния в пространстве признаков [66–69]. Применяемая в данной работе идея состоит в расширении понятия «наименее общего обобщения» (примером может служить антиунификация логических формул [70, 71]) в направлении структурного представления текстовых абзацев и последующем использовании этой операции для вычисления сходства между состоящими из нескольких предложений вопросами и возможными ответами на них.

Рассматриваемое обобщение абзацев текста основано на операции обобщения предложений [72, 73]. Для предложений результатом операции является множество максимальных (по вложению) общих поддеревьев для соответствующих деревьев разбора. Соответственно, наиболее естественным образом операцию структурного обобщения для абзацев можно определить следующим образом.

Определение 2.2. Представим текстовые абзацы P_1 и P_2 в виде ориентированных графов («чащ разбора») G_1 и G_2. Тогда операция обобщения этих абзацев $P_1 \sqcap P_2$ определяется как $\{H_i\}$ - множество всех максимальных по вложению (с учетом меток на вершинах и ребрах) общих подграфов графов из G_1 и G_2.

Такая операция ассоциативна и коммутативна и может быть применена для обобщения произвольного конечного числа абзацев.

В дополнение к построению обобщений для отдельных предложений предпринимается попытка определить, как несинтаксические связи между словами в предложениях могут быть использованы для вычисления сходства между текстами [13]. Для этого применяются специально построенные формализации дискурсивных теорий, в частности, теории риторических структур [74].

2.3 Применение чащ разбора для нахождения ответов на вопросы

Если мы построили последовательность деревьев разбора для вопроса и для ответа, как мы можем сопоставить их между собой? Существует ряд исследований, посвященных вопросу вычисления попарного сходства между деревьями разбора [58, 76]. Тем не менее, для того чтобы использовать связи внутри абзаца и избежать зависимости от распределения содержания по нескольким предложениям ответа, будем рассматривать абзац в целом (т.е. чащу разбора этого абзаца), а не просто отдельные предложения, входящие в этот абзац. В данной концепции для определения того, насколько удачным является ответ на вопрос, достаточно сопоставить чащи ответа и вопроса [42,43,45,47].

2.3.1 Расширенные группы

Для построения структуры абзаца синтаксические отношения, зафиксированные в деревьях разбора, дополним с помощью несинтаксических связей. В качестве таких связей в данной работе используются:

- Кореферентные и таксономические связи:
  - ✓ анафора

- ✓ «та же сущность»
- ✓ «частный случай»
- ✓ «более общий случай» и т.д.

- Связи, полученные с помощью применения дискурсивных теорий (см. раздел 2.3.3).

Используя несинтаксические связи, мы можем расширить понятие синтаксической группы на случай нескольких предложений. При поиске сходства между отдельными предложениями сопоставляются именные, глагольные группы и другие виды групп, фигурирующие в предложениях. Несинтаксические связи между вершинами деревьев разбора позволяют объединять несколько групп из разных предложений или из одного предложения между собой. Таким образом, мы можем расширить понятие группы, допустив включение в группу одной или нескольких несинтаксических связей. Такие связи при обходе группы условно позволяют «перескакивать» с одного дерева разбора на другое. В данной работе рассматриваются следующие типы групп:

- Синтаксические, или регулярные группы;
- Группы, включающие кореферентные (см., например, [77]) и таксономические связи. Для удобства будем называть их *чащевыми* группами.
- RST-группы. Две группы (каждая из них может быть и чащевой, и синтаксической), соединенные RST-отношением.
- CA-группы. Здесь возможны два случая:
  - ✓ Синтаксическая или обычная группа с выделенным в ней коммуникативным действием.
  - ✓ Две группы (каждая из них может быть и чащевой, и синтаксической), объединенные связью между двумя коммуникативными действиями.

Для удобства все объединенные несинтаксическими связями синтаксические группы (чащевые, RST, CA) будем называть *расширенными группами*.

Рассмотрим пример, в котором добавление дополнительных кореферентных связей помогает правильно сопоставить ответ с вопросом:

Ответ 1: … *Tuberculosis is usually a lung disease. It is cured by doctors specializing in pulmonology.*

Ответ 2: … *Tuberculosis is a lung disease… Pulmonology specialist Jones was awarded a prize for curing a special form of disease.*

Запрос: *Which specialist doctor should treat my tuberculosis?*

В обоих случаях тексты содержат ключевые слова из вопроса. Но настоящим ответом является только первый текст. Понять это помогает установление связи *Tuberculosis → disease → is cured by doctors pulmonologists*.

2.3.2 Различные подходы к выявлению сходства между текстовыми абзацами

Существуют различные подходы к оценке сходства между двумя абзацами текста (в рассматриваемых приложениях – вопросом и ответом):

- Применение ключевых слов: базовый подход, в котором тексты представляются в виде «мешка слов», а затем вычисляется набор общих ключевых слов / N-грамм и их частот [69].
- Попарное сравнение предложений: применяются синтаксические обобщения для каждой пары предложений, полученные результаты суммируются [73, 43].
- Попарное сопоставление абзацев текста [53, 73, 43].

Первый подход наиболее характерен для промышленного применения в современной компьютерной лингвистике. Второй

подход был использован, например, в [73]. Ко второму подходу также относятся применение ядер деревьев разбора [76,79] и ядер последовательностей деревьев [79] в алгоритмах классификации типа Метода Опорных Векторов (SVM) [81].

Рассмотрим и сравним перечисленные выше подходы на примере пары коротких текстов (статей). Первый текст можно рассматривать в качестве поискового запроса (причем он необязательно должен быть сформулирован в виде предложения в вопросительной форме), а второй текст – как потенциальный ответ на него. При этом необходимо помнить, что релевантный ответ должен быть тесно связанным с запросом текстом, который в то же время не является копией запроса или его фрагмента.

Примечание. "^" в следующем примере и далее означает операцию обобщения двух абзацев. При описании деревьев разбора используется стандартная нотация, принятая для деревьев составляющих: **[…]** обозначает синтаксическую группу, NN, JJ, NP и т.д. – части речи и типы групп (существительное, прилагательное, именная группа и т.д.), ***** используется для обозначения произвольных вершин дерева. "Communicative action" обозначает коммуникативное действие, <leads to> – связь между коммуникативными действиями, "RST-evidence" – тип риторической связи (см. раздел 1.4.3.2).

"Iran refuses to accept the UN proposal to end the dispute over work on nuclear weapons",
"UN nuclear watchdog passes a resolution condemning Iran for developing a second uranium enrichment site in secret",
"A recent IAEA report presented diagrams that suggested Iran was secretly working on nuclear weapons",
"Iran envoy says its nuclear development is for peaceful purpose, and the material evidence against it has been fabricated by the US",

Λ

“UN passes a resolution condemning the work of Iran on nuclear weapons, in spite of Iran claims that its nuclear research is for peaceful purpose”,
“Envoy of Iran to IAEA proceeds with the dispute over its nuclear program and develops an enrichment site in secret”,
“Iran confirms that the evidence of its nuclear weapons program is fabricated by the US and proceeds with the second uranium enrichment site”

Список общих ключевых слов позволяет определить, что оба документа относятся к ядерной программе Ирана, однако понять на его основе что-то более конкретное весьма затруднительно.

Iran, UN, proposal, dispute, nuclear, weapons, passes, resolution, developing, enrichment, site, secret, condemning, second, uranium

Попарное обобщение предложений дает чуть более полную картину.

[NN-work IN-* IN-on JJ-nuclear NNS-weapons],

[DT-the NN-dispute IN-over JJ-nuclear NNS-*],

[VBZ-passes DT-a NN-resolution],

[VBG-condemning NNP-iran IN-*],

[VBG-developing DT-* NN-enrichment NN-site IN-in NN-secret],

[DT-* JJ-second NN-uranium NN-enrichment NN-site],

[VBZ-is IN-for JJ-peaceful NN-purpose],

[DT-the NN-evidence IN-* PRP-it],

[VBN-* VBN-fabricated IN-by DT-the NNP-us]

Обобщение с помощью чащ разбора дает существенно более детальную картину, чем результаты, полученные с помощью первых двух подходов. См. также рисунок 2.1.

[NN-Iran VBG-developing DT-* NN-*enrichment* NN-site IN-in NN-secret]

[NN-*generalization*-<UN/nuclear watchdog> * VB-pass NN-resolution VBG condemning NN- Iran]

[NN-*generalization*-<Iran/envoy of Iran> *Communicative_action* DT-the NN-dispute IN-over JJ-nuclear NNS-*]

[Communicative_action – NN-work IN-of NN-Iran IN-on JJ-nuclear NNS-weapons]

[NN-generalization <Iran/envoy to UN> *Communicative_action* NN-Iran NN-nuclear NN-* VBZ-is IN-for JJ-peaceful NN-purpose],

[*Communicative_actio*n – NN-*generalization* <work/develop> IN-of NN-Iran IN-on JJ-nuclear NNS-weapons]

[NN-*generalization* <Iran/envoy to UN> *Communicative_action* NN-evidence IN-against NN Iran NN-nuclear VBN-fabricated IN-by DT-the NNP-us]

NN-Iran JJ-nuclear NN-weapon NN-* – *RST-evidence* – VBN-fabricated IN-by DT-the NNP-US

condemn^proceed [enrichment site] <leads to> suggest^condemn [work Iran nuclear weapon]

2.3.3 Несинтаксические связи, получаемые из дискурсивных теорий

Для получения дополнительных несинтаксических связей были использованы и (частично) реализованы в виде программных компонент методы следующих дискурсивных теорий, описывающих отношения внутри абзаца:

- Теория риторических структур (Rhetorical Structure Theory, сокр. RST) [74];
- Теория речевых актов (Speech Act Theory, сокр. SpAcT) [82].

Хотя обе эти теории построены на психологических наблюдениях и имеют в основном невычислительный характер, для них были построены конкретные вычислительные реализации [73]. Для RST из текста извлекаются RST-отношения (риторические отношения). В случае SpAcT для нахождения связей используется

словарь так называемых коммуникативных действий (*communicative actions*) [72].

2.3.3.1 Пример использования риторической структуры

Рассмотрим представленный на рисунке 2.1 пример обобщения на основе риторического отношения «evidence» (доказательство) [82]. Это соотношение имеет место между синтаксическими группами (перед группами указана их роль в риторическом отношении) *«Доказательство-чего [Iran's nuclear weapon program]»* и *«что-происходит-с-доказательством [Fabricated by USA]»*, а также между группами «*свидетельство-чего [against Iran's nuclear development]»* и «*что-происходит-с-доказательством [Fabricated by the USA]»*.

Нужно отметить, что в последнем случае необходимо объединить (путем разрешения анафоры) группу *«its nuclear development»* с группой *«evidence against it»,* чтобы получить *«evidence against its nuclear development»*. Анафорой в данном случае является связь *«it – development»*. *«Evidence»* удаляется из фразы, поскольку это индикатор риторического отношения. Чтобы получить итоговую фразу, необходимо разрешить еще одну анафору: *«its – Iran»*.

После обобщения двух групп, построенных на базе риторического отношения RST-evidence, мы получаем RST-группу «*Iran nuclear NNP – RST-evidence – fabricated by USA»*.

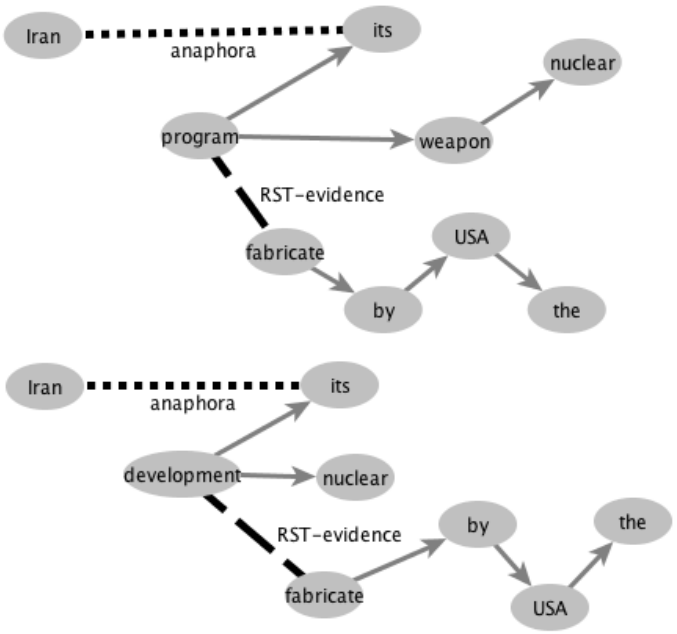

Рис. 2.1. Пример обобщения на основе риторического отношения RST-evidence

2.3.3.2 Обобщение расширенных групп, использующих коммуникативные действия

Инструментарий глаголов – коммуникативных действий используется авторами текстов, для того чтобы показать структуру диалога или конфликта [82]. Поэтому добавление в чащу самих коммуникативных действий и связей, устанавливаемых между ними, позволяет отыскивать неявное сходство между текстами. При выполнении операции обобщения в этом случае применяются следующие правила:

1. Одно коммуникативное действие (глагол) и его субъект (подчиненную группу) из чащи T_1 можно обобщить с другим коммуникативным действием (глаголом) и его субъектом из чащи T_2. Дуга между коммуникативными действиями в этом обобщении не участвует.
2. Пару коммуникативных действий с их субъектами можно обобщить с другой парой коммуникативных действий и их

субъектами из второй чащи. Связь между коммуникативными действиями включается в результат обобщения. Пример такого обобщения приведен на рисунке 2.2.

3. При обобщении двух групп, построенных для коммуникативных действий, в первую очередь обобщаются их субъекты, затем – сами коммуникативные действия. Результат обобщения коммуникативных действий «прикрепляется» к результату обобщения их субъектов, представляющему собой множество максимальных общих поддеревьев. При этом сами коммуникативные действия всегда можно обобщить, но если результат обобщения субъектов является пустым множеством, то и соответствующие им расширенные группы тоже не обобщаются.

2.3.3.3 Пример использования коммуникативных действий

В примере, приведенном на рисунке 2.2, мы имеем совпадающие коммуникативные действия с практически не совпадающими субъектами:

condemn [Iran for developing second enrichment site in secret]
vs
condemn [the work of Iran on nuclear weapon] ,

а также несовпадающие коммуникативные действия с очень похожими субъектами:

suggest [Iran was secretly working on nuclear weapons]
vs
condemn [the work of Iran on nuclear weapon]

Результатом обобщения в первом случае будет пустое множество, поскольку субъекты не обобщаются (см. правило 3). Во втором случае мы получим *suggest^condemn* [*work Iran nuclear weapon*].

Теперь, используя полученные результаты, попробуем обобщить приведенные выше пары коммуникативных действий между собой:

condemn [second uranium enrichment site] ↔ condemn [the work of Iran on nuclear weapon]
↓ *communicative action arcs* ↓
suggest [Iran is secretly working on nuclear weapon] ↔ proceed [develop an enrichment site in secret]

Такое обобщение дает пустое множество, поскольку, как было показано выше, *condemn [Iran for developing second enrichment site in secret]* и *condemn [the work of Iran on nuclear weapon]* не обобщаются.

T_1 T_2

condemn [second uranium enrichment site] ↔ proceed [develop an enrichment site in secret]
↓ *communicative action arcs* ↓
suggest [Iran is secretly working on nuclear weapon] ↔ condemn [the work of Iran on nuclear weapon]

Здесь результатом будет *condemn^proceed* [*enrichment site*] <*leads to*> *suggest^condemn* [*work Iran nuclear weapon*].

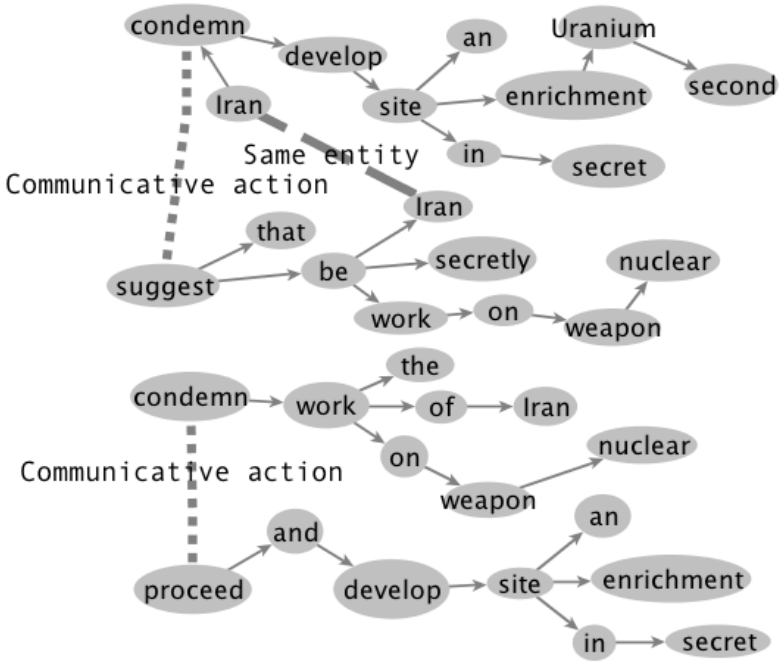

Рис. 2.2. Пример обобщения пар коммуникативных действий и их субъектов

2.4 Вычисление обобщения чащ разбора

Для нахождения обобщения двух чащ используется операция пересечения. Как уже было отмечено выше, она определяется как нахождение всех наибольших общих «подчащ» для двух чащ. В общем случае чаща рассматривается как граф, а пересечение включает в себя все максимальные (по вложению) общие подграфы [44].

Для реализации этой операции мы применили известный метод сведения проблемы нахождения общего подграфа к проблеме нахождения максимальных клик [135], использующий модульное произведение специального вида. Основная разница с традиционным модульным произведением заключается в том, что вместо требования по совпадению меток для пересекаемых ребер применяется ограничение на непустое обобщение этих ребер.

Определение 2.3. Подмножество вершин графа *G* называется *кликой*, если все его вершины попарно смежны.

Клика называется *максимальной*, если на если она не является подмножеством другой клики, и *наибольшей*, если она содержит наибольшее число вершин.

Рассмотрим модульное произведение чащ подробнее. Пусть $G_1 = (V_1, E_1, \alpha_1, L_1)$ и $G_2 = (V_2, E_2, \alpha_{21}, L_2)$ – чащи с вершинами V и ребрами E, где $\alpha : V \to L$ – функция, ставящая метки в соответствие ребрам, а L – конечное непустое множество меток для вершин и ребер. Модульное произведение чащ разбора $H_e = G_1 \circ G_2$ включает множество вершин $V_H = E_1 \circ E_2$, в котором все пары ребер (e_i, e_j), $1 \le i \le |E_1|$ и $1 \le j \le |E_2|$, должны иметь непустое обобщение соответствующих им меток. Помимо этого, данные пары ребер должны иметь непустое обобщение для меток вершин. Пусть

$e_i = (u_1, v_1, l_1)$ и $e_j = (u_2, v_2, l_2)$. Условия выполнены, если $l_1 \cap l_2 \neq \varnothing$, $\alpha_1(v_1) \cap \alpha_2(v_2) \neq \varnothing$ и $\alpha_1(u_1) \cap \alpha_2(u_2) \neq \varnothing$.

Между вершинами $e_H, f_H \in V_H$, где $e_H = (e_1, e_2)$ и $f_H = (f_1, f_2)$ может существовать ребро, если ребра для этой пары не совпадают: $e_1 \neq f_1$ and $e_2 \neq f_2$. Также необходимо выполнение одного из следующих условий:

- e_1, f_1 в G_1 соединены посредством вершины с меткой, которая дает непустое обобщение с меткой вершины, общей для e_2, f_2 в G_2: $\{vertices\ for\ e_1, f_1\} \cap \{vertices\ for\ e_2, f_2\} \neq \varnothing$.
- e_1, f_1 и e_2, f_2 не являются смежными в G_1 и в G_2 соответственно.

Для того чтобы получить общий подграф для G_1 и G_2, для каждой пары ребер в G_1 и G_2 (пара вершин в H_e) из этого подграфа должно существовать обобщение на метках со всеми остальными парами ребер в G_1 и G_2, которые формируют общий подграф. Таким образом, клика в H_e соответствует общим подграфам в G_1 и G_2.

После нахождения всех максимальных клик для всех пар (например, представляющих вопрос и ответы) мы берем все результаты и ранжируем их в соответствии с размером клик. При таком подходе, чем больше пар ребер содержит результат обобщения, тем более релевантным он является.

2.5 Алгоритм вычисления приближенного обобщения чащ разбора

2.5.1 Проекции на чащах

Для того чтобы оценить структурное сходство двух текстовых абзацев, нам необходимо выполнить операцию обобщения на соответствующих им чащах разбора. Воспользуемся приведенными

выше определениями из теории решеток и узорных структур, для того чтобы задать операцию обобщения.

Если рассматривать абзацы как *объекты*, а чащи разбора как их *описания*, то операция *обобщения* или сходства – это полурешеточная *операция пересечения*. Далее, если представить чащу в виде графа, то, как мы уже видели выше, пересечение двух чащ наиболее естественным образом определяется как *множество максимальных общих подграфов* для соответствующих им графов. Выполнение данной операции является NP-трудной задачей [91], поэтому для эффективного вычисления с сохранением свойств операции мы можем воспользоваться введенным выше механизмом *проекций*.

Определение допускает существование большого числа способов задания проекции. В нашем случае наиболее естественно будет воспользоваться лингвистическими свойствами исходной структуры. Зададим проекцию чащи как *множество всех максимальных по вложению синтаксических и расширенных групп*, вычисленных для данного абзаца. Со структурной точки зрения такая проекция – это максимальные по вложению поддеревья графа с дополнительными свойствами. Пример проекций для двух абзацев приведен в разделе 2.3.2.

Операция пересечения двух групп определяется внутри каждого типа групп [73]. Пересечение на проекциях заключается в попарном пересечении групп для каждого типа из двух множеств и выборе наибольших по вложению подгрупп. Работа с проекциями позволяет добиться экономии по сложности (переход к работе с деревьями) без значимого ущерба для качества результата (группы учитывают все необходимые лингвистические связи внутри абзаца).

2.5.2 Построение множества расширенных групп

Рассмотрим подробнее алгоритм нахождения заданной выше проекции, т.е. множества расширенных групп, для чащи разбора.

Для каждого предложения *S* в абзаце *P*:

1. Сформировать список предшествующих предложений в абзаце S_{prev}
2. Для каждого слова в текущем предложении:

2.1 Если это местоимение (*pronoun)*: с помощью разрешения анафоры найти все существительные и именные группы в S_{prev}, которые связаны с данным словом отношением «та же сущность».

2.2 Если это существительное (*noun)*: найти все существительные и именные группы в S_{prev}, которые связаны с данным словом:

- ✓ отношением «та же сущность» (через разрешение анафоры)
- ✓ отношением синонимии
- ✓ отношением «более общий случай»
- ✓ отношением «частный случай»
- ✓ имеют общую родительскую (связанную отношением «более общий случай») сущность

2.3 Если это глагол (*verb*):

2.3.1 Если он выражает собой коммуникативное действие (*communicative action*):

2.3.1.1 Сформировать группу $VBCA_{phrase}$, включающую в себя глагольную группу данного слова VB_{phrase} .

2.3.1.2 Найти предыдущее коммуникативное действие с его субъектом $VBCA_{phrase0}$ в $S_{prev.}$

2.3.1.3 Построить расширенную группу [$VBCA_{phrase0}$, $VBCA_{phrase}$].

2.3.2 Если он указывает на риторическое отношение (*RST relation*):

2.3.2.1 Сформировать из фраз – субъектов отношения расширенную группу [$VBRST_{phrase1}$, $VBRST_{phrase2}$] ; фраза $VBRST_{phrase1}$ относится к S_{prev}.

2.5.3 Обобщение чащ на проекциях

Для того чтобы вычислить сходство для двух абзацев с использованием проекций, необходимо:

1. Выполнить их фрагментацию и извлечь все синтаксические группы из каждого предложения.
2. Найти дискурсивные связи внутри абзаца.
3. Используя дискурсивные связи, построить на основе синтаксических групп расширенные группы.
4. Провести обобщение для каждого из четырех типов (см. раздел 2.3.1) групп, заключающееся в поиске множества максимальных общих подгрупп для каждой пары групп одного и того же типа.
5. Полученные на уровне групп обобщения можно интерпретировать как набор путей в результирующих общих поддеревьях [73].

2.6 Эксперименты по поиску с использованием сходства между абзацами

2.6.1 Схема эксперимента

Попробуем оценить, как обобщение чащ разбора может улучшить поиск в ситуации, когда сформулированный в виде поискового запроса вопрос и потенциальный ответ на него представляют собой текстовые абзацы. Количественным результатом оценки является точность в процентах, вычисленная как среднее по

100 поисковым запросам. В данной работе использовались параметры проведения эксперимента, описанные в [73].

Оценка основана на повторном ранжировании поисковых результатов, полученных с помощью API поисковой системы Bing, осуществляемом на базе меры сходства чащ разбора. Сходство определяется как общее число вершин в наибольшем общем подграфе для двух чащ. Приближенное значение этой оценки было получено путем подсчета количества слов в наибольших общих подгруппах с учетом весов для частей речи [73].

Полученные результаты приведены в таблице. Для оценки были смоделированы следующие ситуации в трех прикладных областях:

- Выдача рекомендаций по товарам: агент рекомендательной системы читает чаты о продуктах и находит в сети релевантную информацию о каждом продукте.
- Выдача рекомендаций по путешествиям: агент рекомендательной системы читает чаты с описанием путешествий и находит в сети релевантную информацию об отелях, курортах и т.д.
- Выдача рекомендаций в социальных сетях (на примере Facebook): агент рекомендательной системы читает чаты и записи на «стене» пользователей социальной сети и отбирает информацию, которая может быть интересна друзьям этих пользователей.

В каждой из этих областей на основе Интернет-источников выбирался кусок текста, далее формировался запрос, а затем

осуществлялась фильтрация результатов поиска, полученных с помощью API поисковой системы Bing.

2.6.2 Результаты экспериментов

Таблица 2.1. Оценка релевантности поиска по точности на первых 10, %

Тип запроса	Сложность запроса	Исходный поиск в Bing	Поиск с использованием обобщений для отдельных предложений	Поиск с помощью чащ, построенных на **фрагментах**	Поиск с помощью чащ, построенных на **оригинальных абзацах**	Поиск с использованием обобщения чащ на **графах**
Поиск рекомендаций по товарам	1 составное предложение	62.3	69.1	72.4	72.9	73.3
	2 предложения	61.5	70.5	71.9	72.8	71.6
	3 предложения	59.9	66.2	72.0	73.4	71.4
	4 предложения	60.4	66	68.5	69.2	66.7
Поиск рекомендаций по путешествиям	1 составное предложение	64.8	68	72.6	74.7	74.2
	2 предложения	60.6	65.8	73.1	76.9	73.5
	3 предложения	62.3	66.1	70.9	70.8	72.9
	4 предложения	58.7	65.9	72.5	73.9	71.7
Поиск рекомендаций контента на Facebook	1 составное предложение	54.5	63.2	65.3	68.1	67.2
	2 предложения	52.3	60.9	62.1	63.7	63.9
	3 предложения	49.7	57	61.7	63.0	61.9
	4 предложения	50.9	58.3	62.0	64.6	62.7
Средние показатели		**58.15**	**64.75**	**68.75**	**70.33**	**69.25**

Точность исходной поисковой выдачи на первых 10 результатах составила 58,2%, применение операции обобщения на уровне предложений дало улучшение в 6,5%. Использование проекций чащ для выдаваемых поисковой системой фрагментов (сниппетов) позволило увеличить точность еще на 4%. Применение полного вычисления обобщения на графах для фрагментов дало прибавку в

0.5%. Наконец, применение проекций, но уже на чащах, построенных для абзацев, которые были извлечены непосредственно из оригинальных документов, дало прибавку еще в 1,5% относительно вычислений на проекциях для фрагментов. Нетрудно видеть, что с ростом сложности запроса увеличивался эффект от применения технологии обобщения. Другим важным выводом является незначительная потеря в точности при существенном выигрыше в скорости за счет использования проекций.

2.7 Оценка вычислительной сложности

Операция обобщения на деревьях разбора и чащах разбора определяется как нахождение всех наибольших общих поддеревьев и подчащ соответственно. Хотя для деревьев эта проблема решается за *O(N),* для графа общего вида она является NP-трудной [91].

Один из подходов к обучению на деревьях разбора основан на так называемых ядрах, определенных для дерева (*tree kernel*). Авторы этого подхода предлагают технику, ориентированную специально на деревья разбора, уменьшая тем самым размерность пространства всех возможных поддеревьев. Существует несколько специальных разновидностей ядер, направленных на более эффективную обработку деревьев. Частичные ядра (*partial tree kernels*) задают правила частичного соответствия, игнорирующие некоторые дочерние узлы [76]. Ядра последовательностей на деревьях (*tree sequence kernels*) используют в качестве подструктуры не просто поддеревья, а последовательности поддеревьев [92].

Подход, основанный на сопоставлении синтаксических групп для предложений и расширенных групп для чащ разбора, с вычислительной точки зрения оказывается гораздо более эффективным, чем подходы, использующие ядра на графах разного

вида, включая деревья. Вместо того чтобы рассматривать пространство всех возможных подграфов, рассматриваются пути в деревьях и графах, которые соответствуют синтаксическим и расширенным группам.

Для того чтобы оценить сложность обобщения двух чащ разбора, рассмотрим абзац, состоящий из 5 предложений, каждое из которых имеет длину в 15 слов. В таких чащах в среднем содержится 10 синтаксических групп в каждом предложении и 10 дуг между предложениями, которые дают нам до 40 расширенных групп. Поэтому для сопоставления таких чащ разбора необходимо попарно обобщить около 50 синтаксических групп и 40 расширенных групп из одной чащи с таким же множеством групп для другой. С учетом обобщения отдельных существительных и глагольных групп это составляет порядка 2 * 45 * 45 обобщений, сопровождаемых проверкой вхождения результатов друг в друга. Каждое обобщение состоит не более чем из 12 сравнений строк, если принять средний размер группы за 5 слов. Следовательно, в среднем обобщение двух чащ включает в себя 2 * 45 * 45 * 12 * 5 операций. Так как сравнение строк занимает несколько микросекунд, обобщение занимает в среднем 100 миллисекунд без использования индекса. Однако в промышленной поисковой системе, где группы хранятся в обратном индексе, операция обобщения может быть выполнена за фиксированное время, не зависящее от размера индекса [93].

2.8 Кластеризация результатов поиска

2.8.1 Решетка замкнутых описаний на чащах

Обработка результатов поиска и их интерпретация – одно из важнейших направлений в промышленном информационном поиске. Проблема отображения результатов часто сводится к их

ранжированию по одному числовому показателю – релевантности. В этом случае результаты выводятся последовательно в соответствии с этим значением. Однако в реальных системах ранжирование производится не только не только по релевантности, но и по месту, времени, ожидаемому доходу от результатов поиска и другим параметрам.

Также существуют и альтернативные варианты отображения результатов поиска, использующие различные виды кластеризации [137, 138]. На практике, как правило, используется комбинация двух подходов: сначала результаты ранжируются по релевантности и из них отбираются N лучших. А затем эти результаты тем или иным образом группируются. Основное преимущество кластеризации заключается в том, что похожие или дублирующие друг друга результаты поиска объединяются, так что пользователь может работать с кластерами результатов, а не с отдельными результатами поиска.

Одним из наиболее перспективных методов кластеризации является концептуальная кластеризация, объединяющая объекты в решетку замкнутых множеств. Такая кластеризация удобна, например, когда поисковая выдача содержит результаты из разных источников: новости, документы, картинки. Если речь идет о социальном поиске, то кластеризация позволяет группировать ответы и темы по пользователям и сообществам. Кроме того, решетка автоматически формирует иерархию и позволяет работать на нужном уровне сходства (например, с большими группами не очень похожих результатов или с маленькими группами почти одинаковых результатов).

Простейшим вариантом концептуальной кластеризации является использование решеток понятий [129,130,131,132,133,136]. Недостатком в данном случае является необходимость предварительного задания множества признаков и проведения шкалирования для получения формального контекста. При этом неизбежна частичная потеря или огрубление информации.

Более сложным случаем является построение решетки на основе замкнутых структурных описаний – узорных структур. В этом случае мы сможем полностью использовать краткое текстовое описание результата – поисковый сниппет [50,51].

Весь необходимый аппарат уже был введен выше. Структурным описанием каждого результата будет являться чаща разбора. Решеточная операция пересечения – это операция сходства чащ разбора. Имея данную операцию, для построения самой решетки можно использовать любой стандартный алгоритм, например, AddIntent [29]. Также в главах 1 и 2 были введены проекции узорных структур. Проекция предоставляет нам приближенное структурное описание, а также способ пересечения этих описаний. Использование проекций для чащ позволяет улучшить временную и вычислительную сложность построения решетки: от операций на графах мы переходим к операциям на деревьях.

Важный момент заключается в том, что используемое описание можно расширять. Чаща разбора – это первое «измерение» в описании поискового результата. Помимо этого, можно также добавлять другие измерения, например, временной интервал, для которого актуален данный результат, целевую аудиторию (например, в виде множества) и т.д. С математической точки зрения, для добавления нового измерения необходимо определить коммутативную и ассоциативную операцию сходства на таких описаниях, которая во многих случаях

вводится естественным образом. Например, для множеств это пересечение, для интервалов – объединение. Необходимо также отметить, что кластеризацию можно применять к произвольным наборам коротких текстов, а группировка результатов поисковой выдачи является лишь одним из приложений данного подхода. Результатом применения описываемого подхода к произвольной коллекции коротких текстов будет являться таксономическое представление этой коллекции, учитывающее синтактико-дискурсивное сходство входящих в неё текстов.

2.8.2 Алгоритм кластеризации

2.8.2.1 Кластеризация с использованием полного описания

Алгоритм кластеризации в случае использования обобщения на полном описании выглядит следующим образом:

1. Взять множество текстов (поисковую выдачу) *T*.
2. Для каждого результата $t_i \in T$ построить чащу разбора $p_i \in P$.
3. Используя операцию обобщения чащ разбора в качестве решеточной операции пересечения $\sqcap$, построить узорную решетку $\left(T,\left(P,\sqcap\right),\delta\right)$ для всех текстов с помощью любого стандартного алгоритма (например, AddIntent или Замыкай-По-Одному).
4. Получить иерархические кластеры – узорные понятия решетки.

2.8.2.2 Кластеризация с использованием проекций

При использовании приближенного представления абзацев алгоритм немного модифицируется:

1. Взять множество текстов (поисковую выдачу) *T*.
2. Для каждого результата $t_i \in T$ построить проекцию чащи разбора $\psi\left(p_i\right) \in \psi\left(P\right)$.

3. Используя операцию обобщения проекций в качестве решеточной операции пересечения, построить проекцию узорной решетки $\left(T,\left(P_\psi,\sqcap_\psi\right),\psi\circ\delta\right)$ для всех текстов с помощью любого стандартного алгоритма (например, AddIntent или Замыкай-По-Одному).
4. Получить иерархические кластеры – проекции узорных понятий решетки.

2.8.3 Пример кластеризации с использованием проекций

Рассмотрим 3 новости и построим для них проекцию узорной структуры:

1) *At least 9 people were killed and 43 others wounded in shootings and bomb attacks, including four car bombings, in central and western Iraq on Thursday, the police said. A car bomb parked near the entrance of the local government compound in Anbar's provincial capital of Ramadi, some 110 km west of Baghdad, detonated in the morning near a convoy of vehicles carrying the provincial governor Qassim al-Fahdawi, a provincial police source told Xinhua on condition of anonymity.*

2) *Officials say a car bomb in northeast Baghdad killed four people, while another bombing at a market in the central part of the capital killed at least two and wounded many more. Security officials also say at least two policemen were killed by a suicide car bomb attack in the northern city of Mosul. No group has claimed responsibility for the attacks, which occurred in both Sunni and Shi'ite neighborhoods.*

3) *A car bombing in Damascus has killed at least nine security forces, with aid groups urging the evacuation of civilians trapped in the embattled Syrian town of Qusayr. The Syrian Observatory for Human Rights said on Sunday the explosion, in the east of the capital, appeared to have been carried out by the extremist Al-Nusra Front, which is allied to al-Qaeda, although there was no immediate confirmation. In Lebanon, security sources said two rockets fired from Syria landed in a border area, and Israeli war planes could be heard flying low over several parts of the country.*

Нижнее понятие соответствует наиболее общему описанию и всем объектам, имеющим это описание. В данном случае это пустое множество объектов и все максимальные по вложению группы из трех новостей.

На следующем уровне мы получаем понятия, каждое из которых содержит 1 объект и его описание.

Узорное содержание для первой новости:

[[NP [JJS-least CD-9 NNS-people], NP [CD-43 NNS-others], NP [NNS-shootings CC-and NN-bomb NNS-attacks], NP [NNS-shootings], NP [NN-bomb NNS-attacks], NP [CD-four NN-car NNS-bombings], NP [JJ-central CC-and JJ-western NNP-Iraq], NP [JJ-central], NP [JJ-western NNP-Iraq], NP [NNP-Thursday], NP [DT-the NN-police], NP [DT-A NN-car NN-bomb], NP [DT-the NN-entrance IN-of DT-the JJ-local NN-government NN-compound IN-in NNP-Anbar POS-'s JJ-provincial NN-capital IN-of NNP-Ramadi ,-, DT-some CD-110 NN-km NN-west IN-of NNP-Baghdad], NP [DT-the NN-entrance]…, и т.д.

Узорное содержание для второй новости:

*[[NP [NNS-Officials], NP [DT-a NN-car NN-bomb IN-in JJ-northeast NNP-Baghdad], NP [DT-a NN-car NN-bomb], NP [JJ-northeast NNP-Baghdad], NP [CD-four NNS-people], NP [DT-another NN-bombing IN-at DT-a NN-market IN-in DT-the JJ-central NN-part IN-of DT-the NN-capital], NP [DT-another NN-bombing], NP [DT-a NN-market IN-in DT-the JJ-central NN-part IN-of DT-the NN-capital],…*и т.д.

Узорное содержание для третьей новости:

[[NP [DT-A NN-car NN-bombing IN-in NNP-Damascus], NP [DT-A NN-car NN-bombing], NP [NNP-Damascus], NP [JJS-least CD-nine NN-security NNS-forces], NP [NN-aid NNS-groups VBG-urging DT-the NN-evacuation IN-of NNS-civilians VBN-trapped IN-in DT-the JJ-embattled JJ-Syrian NN-town IN-of NNP-Qusayr], NP [NN-aid NNS-groups], ...)

Понятие верхнего уровня содержит группы, которые являются общими для всех текстов. В данном случае все 3 текста повествуют о взрывах машин возле столиц (*car bombing near capitals*), что выражается фрагментами *[DT-a NN-car NN-bombing], [DT-the NN-*

capital], [VBN-killed], [JJS-least CD- NN-*]*. Символ ‘*’ означает «произвольное слово, относящееся к данной части речи».

Другие группы в данном узорном содержании соответствуют синтаксическим шаблонам: *[IN-of DT-the], [NNS-* IN-* DT-* NN-*]*. На уровне пересечения пар текстов наиболее интересным является понятие, содержащее тексты 1 и 2. Они описывают одно и то же событие, поэтому место происшествия совпадает: *[NN-* NN-* IN-in NNP-**baghdad**]*. В обоих текстах используются одинаковые термины: *[NN-* NN-**bomb** NN-**attack**], [NNS-**attacks**]*, и информация о пострадавших: *[VBD-**wounded**], [VBD-**were** VBN-**killed**], [CD-* NNS-**people**], [CD-**four** NNS-*]*.

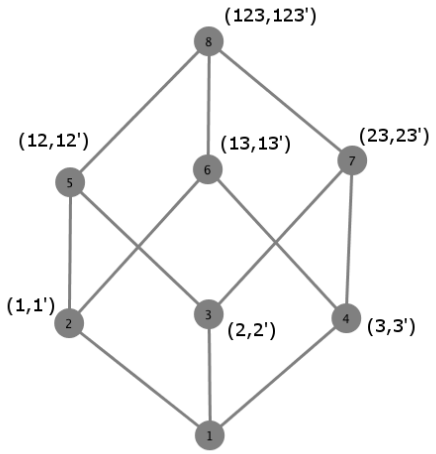

Рис 2.3. Проекция узорной структуры для новостных текстов

2.9 Выводы

В работах [73, 94] было показано, как использование богатого набора лингвистической информации – синтаксических связей между словами – улучшает релевантность поиска. Для того чтобы, помимо синтаксической информации, воспользоваться и дискурсивной, было использовано понятие чащи разбора и предложен способ вычисления сходства между текстами, основанный на обобщении соответствующих им чащ разбора. Также была построена и применена

к задаче повторного ранжирования результатов поиска по ключевым словам технология обобщения чащ разбора, представляемых в виде наборов групп.

Для построения чащ использовались различные виды связей между словами в предложениях: кореферентные связи, таксономические отношения, такие как «быть частным случаем», «быть обобщением» и т.д., а также дискурсивные связи, полученные на базе теории риторических структур и теории речевых актов. Было показано, что если ответ содержится в нескольких предложениях, то применение чащи позволяет повысить релевантность поиска.

Также было показано, что операция сходства или обобщения абзацев текста может быть естественным образом определена с помощью математического аппарата узорных структур. При этом обобщение с использованием общих подграфов точно описывается в терминах пересечения описаний объектов, а синтаксические и расширенные группы соответствуют взятию проекций от исходных описаний.

Традиционное машинное обучение на языковых структурах ограничено работой с формами и частотами ключевых слов. В то же время большинство семантических теорий не являются вычислительными, они моделируют определенный набор отношений между последовательными состояниями. В данной работе была предпринята попытка совместить два подхода: использовать всю информацию, полученную из дерева синтаксического разбора, дополнив ее сведениями из дискурсивных теорий, допускающих вычислительную обработку.

3. Применение ядер для классификации коротких текстов

3.1 Введение

Несмотря на значительные усилия по формулированию полноценной теории, описывающей связь между синтаксисом и семантикой, она все ещё не разработана. Однако конструирование синтаксических признаков для автоматического обучения на синтаксических структурах можно назвать мейнстримом. Одно из решений для работы с такими признаками – построение и вычисление ядер на деревьях синтаксического разбора. Функция ядра (convolution kernel) на деревьях [76] задает пространство признаков, состоящее из возможных типов поддеревьев деревьев разбора, и подсчитывает количество общих подструктур в качестве синтаксической близости между деревьями. Этот подход имеет несколько приложений в различных задачах компьютерной лингвистики, в частности, он используется для извлечения отношений [78, 79], распознавания именованных сущностей [111] и выявления семантических ролей (Semantic Role Labeling) [112], разрешения анафоры на местоимениях [119], классификации вопросов [118] и машинного перевода [120].

Свойство ядер генерировать большие объемы признаков является полезным для быстрого моделирования новых и не очень хорошо изученных лингвистических явлений в обучающих алгоритмах. Однако всегда возможно вручную смоделировать признаки для линейных ядер, для того чтобы добиться высокой точности и хорошей скорости работы, несмотря на то что сложность ядер на деревьях может помешать их применению в реальных приложениях.

Многие обучающие алгоритмы, такие как Метод Опорных Векторов (SVM) [81], могут работать напрямую с ядрами с помощью замены скалярного произведения на конкретную функцию ядра («трюк с ядрами»). Это полезное свойство ядер делает их эффективным решением для моделирования структурных объектов в задачах обработки текстов на естественном языке. Некоторые их этих задач требуют вычисления дискурсивных свойств абзацев, содержащих несколько предложений. Использование попарного сравнения предложений не всегда является хорошим вариантом, поскольку в таком случае мы попадаем в зависимость от того, как информация (синтаксические группы) распределены между предложениями.

Помимо ядер на отдельных деревьях был разработан и подход к построению ядра, базирующегося более чем на одном дереве разбора: ядра для леса деревьев. Однако, как правило, такие ядра использовались не для обработки кусков текста, состоящих из нескольких предложений, а для других задач. Одним из применений леса является задача компенсации ошибок синтаксического разбора [80]. В этом случае для каждого предложения строится лес из n лучших деревьев синтаксического разбора, что дает гораздо более богатый набор признаков по сравнению с одиночным деревом. Это преимущество позволяет ядру для леса не только быть более устойчивым по отношению к ошибкам разбора, но и давать более надежные значения признаков, а также помогает решить проблему разреженности данных, которая существует в традиционных ядерных функциях на деревьях.

В работах [115, 116], освещавших задачу поиска ответов на сложные вопросы, лес деревьев применялся для обучения на текстах из нескольких предложений. Однако связи между предложениями в

этом случае не строились и не учитывались. Кроме того, обучение производилось на ответах на все вопросы, а не только на данный (как описано в экспериментах ниже), что представляется не вполне оправданным.

В нашем исследовании [46] мы формируем лес деревьев для нескольких взаимосвязанных предложений, а не для одного. В поиске ответов на вопросы, когда вопрос и ответ состоят из одного предложения, классические методы (как раз и ориентированные на одиночные предложения) дают хорошие результаты. Однако при решении задачи обучения на текстах, состоящих из нескольких предложений, необходимы структуры, описывающие взаимоотношения внутри абзаца. Мы демонстрируем, что в определенных случаях использование дискурсивной информации для обучения и рассмотрение абзацев и связей внутри абзаца дает преимущество по сравнению со стандартными методами.

3.2 Пример расширения деревьев разбора

При анализе коротких текстов попарного сравнения предложений недостаточно для полноценного обучения дискурсивным свойствам текста. Этот факт связан с существованием различных способов распределения информации по нескольким предложениям и различных дискурсивных структур, которыми может быть наделен текст и которые необходимо учитывать.

Рассмотрим пример, в котором короткие фрагменты текста принадлежат двум классам:

- Налоговые обязательства владельца, сдающего свой офис организации или бизнесмену.
- Налоговые обязательства бизнесмена или организации, арендующей офис у владельца.

I rent an office space. This office is for my business. I can deduct office rental expense from my business profit to calculate net income.

To run my business, I have to rent an office. The net business profit is calculated as follows. Rental expense needs to be subtracted from revenue.

To store goods for my retail business I rent some space. When I calculate the net income, I take revenue and subtract business expenses such as office rent.

I rent out a first floor unit of my house to a travel business. I need to add the rental income to my profit. However, when I repair my house, I can deduct the repair expense from my rental income.

I receive rental income from my office. I have to claim it as a profit in my tax forms. I need to add my rental income to my profits, but subtract rental expenses such as repair from it.

I advertised my property as a business rental. Advertisement and repair expenses can be subtracted from the rental income. Remaining rental income needs to be added to my profit and be reported as taxable profit.

Во-первых, отметим, что анализ с помощью ключевых слов не помогает отделить первые три абзаца от последних трех. Все они содержат ключевые слова *rental/office/income/profit/add/subtract*.

Анализ, основанный на использовании синтаксических групп, в данном случае оказывается бесполезным по аналогичной причине.

Попарное сравнение предложений также не решает поставленную проблему.

Использование кореферентных связей между предложениями (разрешение анафоры) помогает, но лишь частично: все эти предложения содержат местоимение *‘I’* и отсылки к нему. В связи с этим очевидно, что необходимо использование дополнительных связей между предложениями. Источником таких связей могут

служить риторические структуры, уже использовавшиеся ранее. Структуры, описывающие фразы *renting for yourself and deducting from total income* и *renting to someone and adding to income*, затрагивают несколько предложений. Второе условие *adding/subtracting incomes* связано риторическим отношением *elaboration* с первым аргументом для *landlord/tenant*. Это риторическое отношение может связывать блоки, расположенные внутри предложения, в предложениях, идущих друг за другом, и даже в предложениях, между которыми есть другие предложения, например, блоки в 1 и 3 предложениях.

На рисунке 3.1 показаны деревья зависимостей и кореферентные связи для предложений первого текста. Есть несколько способов, с помощью которых можно соединить вершины разных деревьев: мы выбрали риторическое отношение *elaboration*, которое помогает нам сформировать структуру *rent-office-space – for-my-business – deduct-rental-expense*, являющуюся базой для нашей классификации. Мы использовали Stanford Core NLP, модуль для работы с кореферентными связями [77], включающий средство для визуализации, для того чтобы построить связи, изображенные на рисунках 3.1 и 3.2.

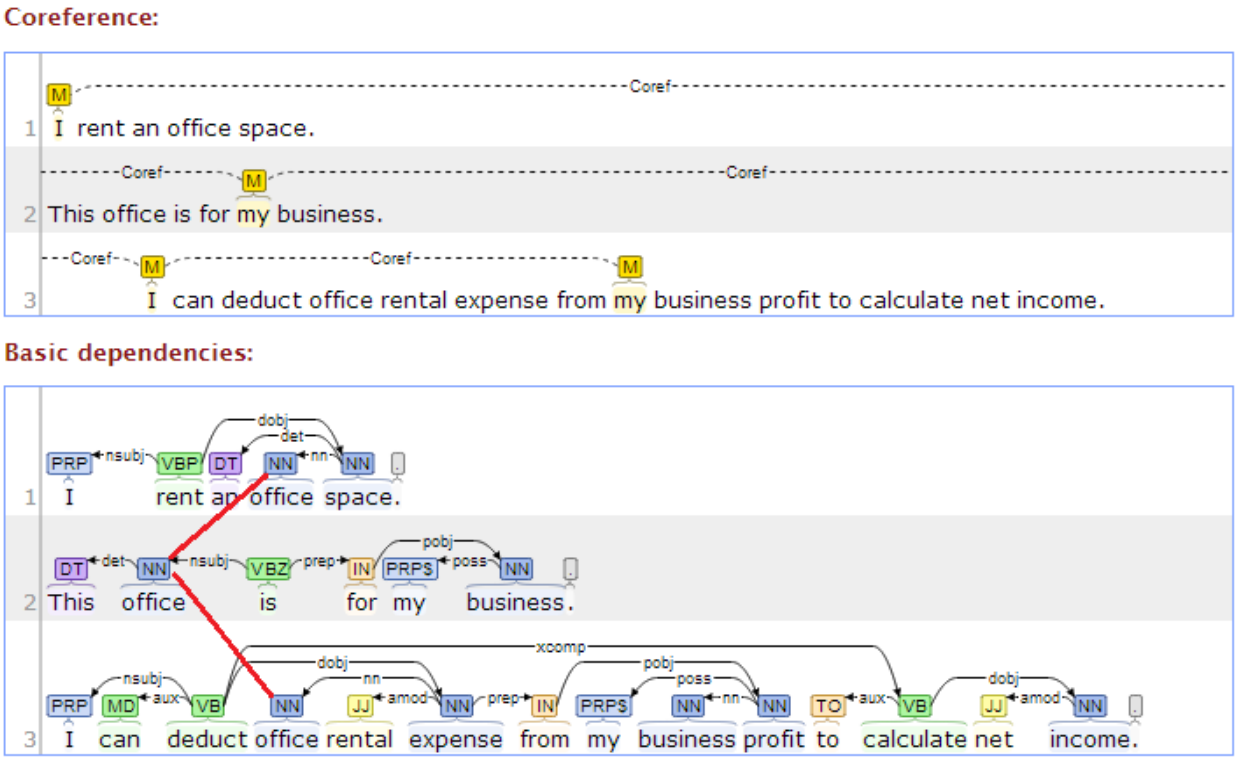

Рис.3.1. Кореферентные связи и множество деревьев зависимостей для первого текста.

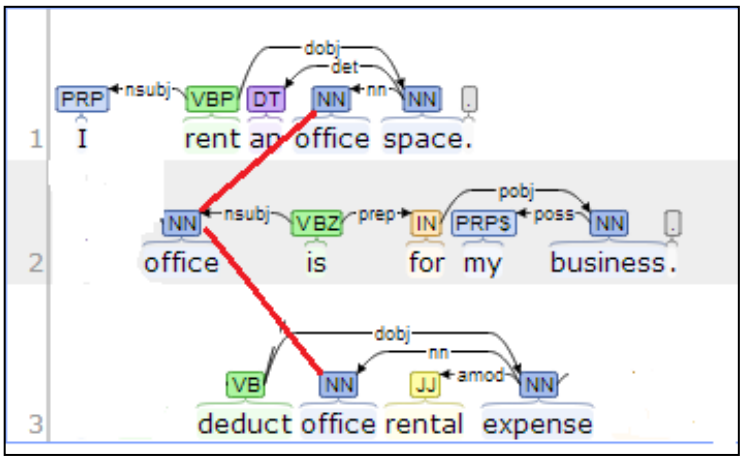

Рис. 3.2. Расширенное дерево, включающее фрагменты трех предложений

На рисунке 3.2 изображено итоговое дерево с корнем ‘*I*’ из первого предложения. Оно полностью включает в себя первое дерево, глагольную группу из второго предложения и глагольную группу из третьего предложения в соответствии с риторическим отношением *elaboration*. Необходимо отметить, что это расширенное дерево с интуитивной точки зрения может рассматриваться как

представляющее «главную идею» текста в сравнении с остальными текстами в нашем множестве. Поскольку заранее неизвестно, какое именно дерево окажется ключевым, необходимо сформировать все расширенные деревья для текста и затем сопоставить их с деревьями остальных текстов. С точки зрения обучения на деревьях, расширенные деревья могут быть использованы совершенно аналогично обычным деревьям разбора.

3.3 Алгоритм построения расширенных деревьев

Для каждой дуги, соединяющей два дерева разбора, построенные для предложений, мы строим пару расширенных деревьев, делая новый переход по этой дуге (рисунок 3.3).

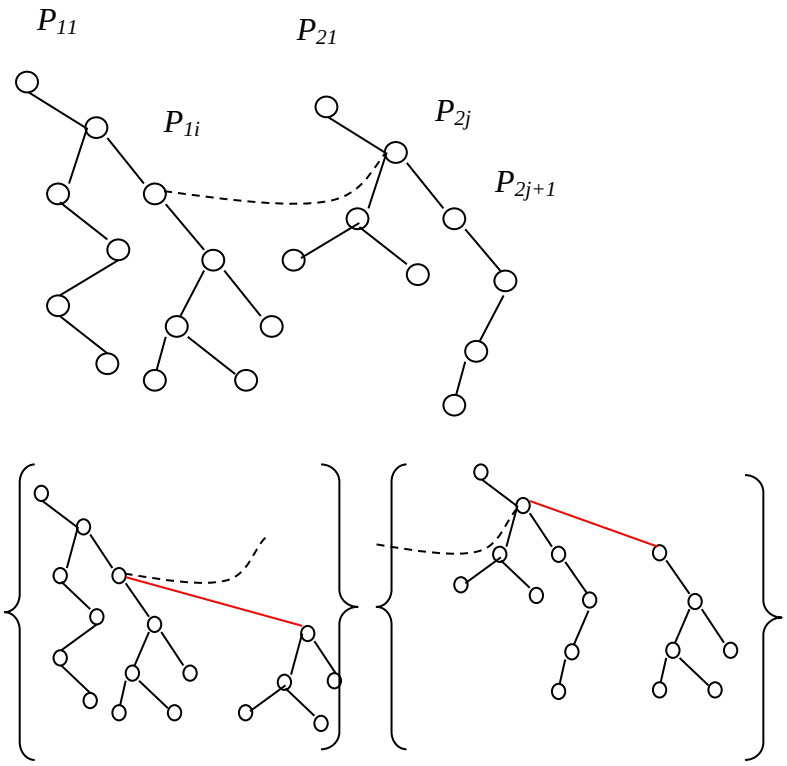

Рис. 3.3. Дуга, которая соединяет два дерева разбора для двух предложений в тексте (верхняя часть), и полученное на её основе множество расширенных деревьев (нижняя часть)

Если у нас есть два дерева разбора P_1 и P_2 для двух предложений абзаца и отношение R_{12}: $P_{1i} \rightarrow P_{2j}$, соответствующее связи между

вершинами P_{1i} и P_{2j}, мы можем сформировать пару расширенных деревьев $P_1 * P_2$:

…,P_{1i-2}, P_{1i-1}, P_{1i}, P_{2j}, P_{2j+1}, P_{2j+2},…

…,P_{2j-2}, P_{2j-1}, P_{2j}, P_{1i}, P_{1i+1}, P_{2i+2},…,

Эти деревья будут использованы для классификации в дополнение к исходным деревьям P_1 и P_2. Необходимо отметить, что исходный порядок следования вершин сохраняется при применении операции ‘*’ (рисунок 3.3).

Ниже приведен алгоритм построения расширенных деревьев на основе множества *T*:

Вход:
1) Множество деревьев разбора *T*.
2) Множество отношений *R*, которое включает в себя отношения R_{ijk} между деревьями T_i and T_j: $T_i \in T$, $T_j \in T$, $R_{ijk} \in R$. Индекс *k* необходимо использовать, поскольку между каждой парой деревьев может существовать несколько связей.

Выход: расширенные деревья *E*.

$E = \varnothing$;
Для каждого дерева i=1:|T|
  Для каждого отношения R_{ijk}, k= 1: |R|, j >= i
    Взять T_j
    Сформировать пару расширенных деревьев $T_i * T_j$
    Для каждого дерева *L* в *E*:
      Проверить, вкладывается ли *L* в $T_i * T_j$
      Если вкладывается, удалить *L* из *E*
      Проверить, вкладывается ли $T_i * T_j$ в *L*
      Если вкладывается, не включать $T_i * T_j$ в *E* и перейти к следующему отношению
    Если ни одно дерево не включает в себя $T_i * T_j$, добавить его в *E*;
Вернуть *E*.

Итоговые деревья не являются корректными деревьями синтаксического разбора, однако формируют адекватное пространство признаков для ядер на деревьях.

Для построения связей между деревьями использовались следующие источники:

1. Кореферентные связи из Stanford NLP [123].
2. Риторические структуры.
3. Коммуникативные действия (для поиска с помощью классификации).

3.4 Оценка вычислительной сложности

Чтобы оценить сложность построения расширенных деревьев, рассмотрим усреднённый случай: 5 предложений в каждом абзаце и 15 слов в каждом предложении. Мы в среднем имеем 10 связей между предложениями, которые дают нам верхнюю границу в 20 расширенных деревьев для двух предложений и 60 для трех. Таким образом, нам необходимо применить обучение для 100 деревьев вместо исходных 5. То есть применение расширенных деревьев дает увеличение входных данных в 20 раз.

Однако большинство маленьких поддеревьев повторяют друг друга и будут сокращены при снижении размерности. Также стоит отметить, что в промышленных поисковых системах, где лингвистические группы хранятся в инвертированном индексе, операция вычисления функции ядра на поддеревьях может выполняться за фиксированное время, вне зависимости от размера индекса [109]. В случае реализации этой операции с помощью технологии *map-reduce*, например, с использованием программного

инструмента Cascading [19], временная сложность становится постоянной и не зависит от числа деревьев [93].

3.5 Эксперименты

3.5.1 Поиск с помощью классификации

Для того чтобы убедиться в том, что использование множества расширенных деревьев дает выигрыш по сравнению с использованием множества обычных деревьев для абзаца, мы провели эксперимент по определению релевантности поиска [46]. Мы применили один и тот же тип ядерной функции для обучения на абзацах, формируя множество деревьев следующими способами:

1. Множество деревьев для предложений абзаца.
2. Все расширенные деревья для предложений абзаца.

Затем мы сравнили результаты классификации, полученные в результате применения обучающего алгоритма, для выбранных вариантов. Важно отметить, что поиск позволяет получить доступ к неограниченному количеству коротких текстов. Во всех экспериментах использовался Bing API.

Поскольку стандартного набора данных для сложных вопросов, состоящих из нескольких предложений, до сих пор не существует, мы составили свой набор для мнений и отзывов о товарах. Задача нахождения ответа на вопрос в данном случае формулируется как нахождение информации в сети, релевантной записи/выражению мнения пользователя в блоге, на форуме или в социальной сети. Мы сгенерировали множество запросов, являющихся текстовыми абзацами, и запустили поисковый механизм Bing API, чтобы найти потенциальные ответы и составить обучающую выборку.

Задача обучения формулируется как классификация множества поисковых результатов по двум классам: релевантные и нерелевантные. Соответствующая обучающая выборка формируется как множество ответов с высоким рейтингом (положительные примеры) и множество ответов с низким рейтингом (отрицательные примеры). Тестовая выборка формируется из оставшегося множества путем случайного выбора. Для каждого результата мы используем его «сниппет», а также соответствующий ему фрагмент текста, извлеченный со страницы. Этот эксперимент базируется на предположении, что верхние (нижние) результаты, выдаваемые Bing, так или иначе релевантны (нерелевантны) исходному запросу, несмотря на то что они могут быть неверно упорядочены.

При проведении данного эксперимента было не столь важно получить наилучшее возможное множество ответов. Основное внимание было сосредоточено на оценке улучшения релевантности, достигаемого за счет использования расширенных деревьев.

Мы поставили задачу подобрать или подготовить набор данных, который удовлетворял бы нескольким требованиям. Во-первых, представлял бы собой естественным образом разбитое на несколько классов множество коротких текстов, имеющих более-менее четко выраженную лингвистическую структуру. Во-вторых, содержал бы достаточное количество реальных описаний каких-либо объектов, имеющих практическую ценность. К сожалению, найти готовый набор, обладающий обоими этими свойствами или хотя бы только первым из них, не удалось. В связи с этим мы решили использовать опыт, накопленный в работах по исследованию и улучшению поиска [43], и использовать в качестве выборки для обучения и тестирования выдаваемые промышленной поисковой системой результаты (первые

N) предварительно подобранных поисковых запросов. В качестве классов наиболее естественным было выбрать классы «релевантны исходному запросу» и «нерелеватны исходному запросу». В этой концепции каждая поисковая выдача соответствует одной выборке. Одинаковая природа всех выборок позволяет усреднить полученные показатели точности и полноты по всем экспериментам. Таким образом, повышение доверия к результатам в нашем случае достигается за счет увеличения числа экспериментов, а не за счет увеличения размера исходной выборки.

Подготовка поисковых запросов происходила в несколько этапов:

1. Отбор названий и коротких (несколько слов) описания продуктов.
2. Поиск расширенных обзоров и мнений о продуктах на основе сформированных описаний.
3. Извлечение из полученных текстов максимальных по вложению и именных и глагольных групп. Этот этап необходим, поскольку оригинальные тексты дают слишком специфические запросы, приводящие к появлению малого числа дублирующих друг друга результатов.

Аналогичные шаги были сделаны для подготовки запросов на базе данных с ресурса Yahoo Answers. В итоге были выбраны порядка 100 запросов для каждой из этих двух областей.

Для классификации результатов поисковой выдачи по каждому из запросов мы использовали следующую схему. Сначала берутся первые 100 (или все, если результатов было меньше) результатов. Далее из этого множества в качестве обучающей выборки выделяются первые и последние 20%, которые рассматриваются как,

соответственно, положительные и отрицательные примеры. Для проведения классификации из остальной части множества случайным образом выбираются K результатов, K вычисляется исходя из соотношения 1 к 4 между тестовой и обучающей выборкой (K = 10, если результатов ровно 100).

В основе такого подхода лежит несколько идей. Во-первых, мы предполагаем, что первые результаты (это подтверждается проведенными исследованиями в области поиска), выдаваемые поисковым движком, так или иначе являются релевантными запросу. Они, разумеется, могут быть неправильно упорядочены, поэтому в нашей схеме порядок не играет роли. Тот факт, что эти примеры не являются «золотым стандартом» и могут быть частично некорректными, также является скорее преимуществом, чем недостатком, поскольку на реальных данных редко удается подобрать идеальную обучающую выборку. Кроме того, такой подход позволяет дополнительно автоматизировать проведение экспериментов, поскольку размечать приходится только тестовую выборку, которая всегда меньше обучающей.

Использование в качестве отрицательных примеров «последних из первых» является вынужденной мерой лишь отчасти. Отрицательные примеры в данном случае, как и положительные, содержат ключевые слова (возможно, не все) из исходного запроса. Однако отличие между ними состоит в том, что в положительных примерах встречаются фразы из исходного запроса, образующие смысловые единицы, и обучение на деревьях как раз призвано уловить это отличие. А использование расширенных деревьев помогает выделить случаи, в которых исходные фразы распределены между несколькими предложениями в тексте.

Также необходимо отметить, что для каждой поисковой выдачи производилось фактически два независимых эксперимента. В одном случае в качестве исходных данных рассматривались так называемые сниппеты (snippets или passages) – короткие фрагменты, обычно отображаемые поисковым движком непосредственно на странице поиска и представляющие собой объединение нескольких наиболее релевантных отрывков текста. В рамках сниппета мы делали из каждого такого отрывка отдельное предложение и объединяли их в один абзац. Во втором случае на базе сниппета и оригинального текста с найденной поисковиком страницы автоматически формировалась краткая выдержка (summary), содержащая наиболее близкие к сниппету предложения со страницы.

Обучение и классификация осуществлялись в автоматическом режиме с использование программного средства SVMLight (http://disi.unitn.it/moschitti/Tree-Kernel.htm [114]). Параметры были рекомендованы автором ПО. Для работы с обычными и расширенными деревьями использовалось представление «лес деревьев» (packed forest). Как уже отмечалось выше, ядро в этом случае вычисляется как нормированная сумма всех функций ядер для каждой пары деревьев леса. Оценка точности и полноты (отнесение результатов к релевантным/нерелевантным) производилась вручную.

Таблица 3.1. Результаты для запросов, связанных с мнением о продуктах. Обучение на текстах со страниц,%

Продукты	*Ядра на обычных деревьях*	*Ядра на расширенных деревьях*
Точность	56,8	**58,7**
Полнота	75,2	**84,6**
F-мера	64,9	**67,5**

Таблица 3.2. Результаты для запросов, связанных с мнением о продуктах. Обучение на поисковых сниппетах,%

Продукты	*Ядра на обычных деревьях*	*Ядра на расширенных деревьях*
Точность	56,3	**63,2**
Полнота	78,4	**83,1**
F-мера	61,7	**67**

Таблица 3.3. Результаты для запросов, сформированных на базе вопросов из *Yahoo Answers*. Обучение на текстах со страниц, %

Yahoo Answers	*Ядра на обычных деревьях*	*Расширенные деревья (только кореферентные связи)*	*Расширенные деревья*
Точность	51,7	50,8	**54,4**
Полнота	73,6	79,2	**83,3**
F-мера	60,1	54,6	**62,8**

Таблица 3.4. Результаты для запросов, сформированных на базе вопросов из *Yahoo Answers*. Обучение на поисковых сниппетах,%

Yahoo Answers	*Ядра на обычных деревьях*	*Расширенные деревья (только кореферентные связи)*	*Расширенные деревья*
Точность	59,5	62,6	**67,9**
Полнота	73,3	74,9	**79**
F-мера	62,5	64,3	**70,7**

Результаты экспериментов, усредненные по всем поисковым запросам, показывают ощутимое улучшение, достигаемое за счет использования расширенных деревьев. На примере Yahoo Answers видно, что добавление только кореферентных связей дает небольшой прирост, тогда как использование и кореферентных связей, и риторических структур позволяет добиться более существенной прибавки. Более существенный прирост полноты по сравнению с точностью объясняется тем, что использование дискурсивной информации позволяет корректно классифицировать как релевантные

тексты, в которых исходные фразы распределены между несколькими предложениям.

Исходные и преобразованные запросы, тестовая выборка, а также подробные результаты классификации доступны на ресурсах http://code.google.com/p/relevance-based-on-parse-trees и https://github.com/bgalitsky/relevance-based-on-parse-trees.

3.5.2 Классификация технических документов

Ещё один эксперимент, в котором проверялся предлагаемый метод – классификация технических документов [48]. В этом случае рассматриваются документы, относящиеся к двум классам:

1. Action-plan (описание оригинальной разработки) - документ, который содержит четкое и хорошо структурированное описание того, как построить конкретную систему в какой-либо области.
2. Meta-document (мета-описание) – документ, объясняющий, как писать документы, относящиеся к первому классу, например, инструкция, учебник, технический стандарт и т.д.

Данная задача важна с практической точки зрения. «Мета-документы», как правило, содержат общедоступную информацию и могут распространяться свободно. Описание же оригинальных разработок является собственностью компаний и не может передаваться и копироваться без их разрешения.

Очевидно, что технические документы, относящиеся к одной области, будут содержать примерно один и тот же набор ключевых слов и словосочетаний. Использование синтаксической информации тоже не дает полной картины, поскольку такого рода тексты обычно написаны стандартизованным языком с использованием коротких связанных друг с другом предложений. В то же время, разумеется,

разделение классов нельзя считать аналитической задачей. Разумеется, описания разработок могут содержать фрагменты мета-описаний (например, как отсылка к стандарту). И наоборот – в мета-документы могут быть включены фрагменты описаний конкретных разработок (в качестве примеров). В связи с этим применение статистического метода обучения, использующего лингвистическую информацию, представляется вполне обоснованным.

Для класса «action-plan» мы сформировали набор данных из 940 оригинальных документов. Для второго класса мы также подобрали набор документов с мета-описаниями на близкие инженерные темы. Эти мета-документы содержали те же ключевые слова, что и оригинальные документы. Затем данные были разбиты на 3 группы для проведения обучения и тестирования по методу кросс-валидации [101].

Таблица 3.5. Результаты классификации технических документов.

Метод	**Точность,%**	**Полнота,%**	**F-мера,%**
*«Ближайшие соседи» (на основе TF*IDF)*	53.9	62	57.67+-0.62
Наивный Байесовский	55.3	59.7	57.42+-0.84
Ядра на синтаксических деревьях	71.4	76.9	74.05+-0.55
Ядра на расширенных деревьях (только анафора)	77.8	81.4	79.56+-0.70
Ядра на расширенных деревьях (только RST)	80.1	80.5	80+-1.03
Ядра на расширенных деревьях (анафора +RST)	**83.3**	**83.6**	**83.45+-0.78**

В качестве альтернативных методов для сравнения использовался метод, основанный на использовании ядер на синтаксических деревьях, а также несколько стандартных классификаторов. В их число вошли метод ближайших соседей и наивный байесовский подход [100,102]. Эти методы использовали только статистику по ключевым словам [98,99].

Альтернативные подходы продемонстрировали достаточно низкое качество. Наилучшим среди них, как и следовало ожидать, оказался метод, использующий информацию о синтаксических связях. Применение анафоры без использования других дискурсивных связей дало прибавку по F-мере относительно этого метода примерно в 5%. Применение риторических связей (опять же изолированно от остальных типов дискурсивных связей) позволило улучшить результат «синтаксического» метода на 6%. Наилучший результат продемонстрировал метод, комбинирующий все типы дискурсивных связей: +9%.

3.6 Выводы

В данной главе было показано, как использование дискурсивной информации позволяет улучшить качество классификации коротких текстов. Эксперименты проводились на задаче поиска с помощью классификации и на задаче классификации технических текстов. Мы провели сравнение двух основных вариантов обучения:

- Обучения на деревьях разбора для отдельных предложений,
- Обучения на деревьях разбора для отдельных предложений, дополненных расширенными деревьями разбора – деревьями, полученными на основе дискурсивных связей между предложениями абзаца.

Было показано, что добавление новых признаков без изменения схемы эксперимента улучшает качество классификации по сравнению с методом, использующим только синтаксические связи. Это улучшение колеблется в диапазоне от 2 до 8% для текстов из нескольких областей, имеющих различную структуру. При этом важно отметить, что это улучшение и внедрение дополнительных признаков не потребовали доработки самого алгоритма обучения на

деревьях. В задаче классификации технических документов, помимо чисто «синтаксического» метода, сравнение проводилось также с классификаторами, использующими статистику по ключевым словам (метод ближайших соседей и метод наивной байесовской классификации). Преимущество нового метода по точности и полноте составило более 15%.

В главе 2 было продемонстрировано, что использование различных дискурсивных связей между предложениями (кореферентные связи, риторические структуры, коммуникативные действия) позволяет добиться улучшения качества поиска в случае, когда ответ содержится в нескольких предложениях. Оказывается, использование дополнительных связей также позволяет улучшить качество классификации коротких текстов. При этом сам алгоритм вычисления ядра не модифицируется.

Построение расширенных деревьев, рассмотренных выше, подразумевает использование проекций. Множество расширенных деревьев абзаца является проекцией чащи разбора. При этом необходимо отметить, что данный вид проекции отличается от проекций, описанных в главе 2, которые применялись для нахождения сходства текстовых абзацев. В главе 2 проекция чащи определялась как множество всех максимальных по вложению подграфов, являющихся деревьями. В текущей главе рассматривался упрощенный вариант данной проекции, подразумевающий в результирующих деревьях не более одной дискурсивных связи. Использование этой проекции допустимо, поскольку оно не приводит к потере информации: все дискурсивные связи чащи попадают в лес деревьев, используемый при обучении. В то же время применение данной проекции позволяет упростить алгоритм построения расширенных

деревьев и снизить вычислительную сложность подготовки данных для обучения и классификации.

Применение ядер для анализа коротких текстов является альтернативой описанному в главе 2 построению узорной структуры на чащах. Ядра позволяют осуществить обучение с учителем, тогда как построение узорной структуры является ни чем иным как иерархической кластеризацией, то есть обучением без учителя.

Стоит отметить, что предложенный подход может быть также применен к проблеме построения запросов и обхода для самих деревьев разбора. Эта задача актуальна в таких системах как Tregex [121], работающих на уровне отдельных предложений. В случае расширения области действий таких систем до абзацев текста за счет внедрения расширенных деревьев полнота систем существенно возрастет, а их зависимость от того, как распределена информация между предложениями, напротив, снизится.

Другим интересным продолжением работы является исследование качества ранжирования результатов, получающегося в результате экспериментов, с помощью стандартных метрик качества, таких как NDCG. В этом случае можно сравнить ранжирование с помощью обычных и расширенных ядер с исходным ранжированием Bing, а также с различными методами, позволяющими переупорядочивать поисковую выдачу на основе синтаксической и дискурсивной структуры результатов.

Также в качестве направления для развития исследования можно отметить внедрение в существующий метод ядер, вычисляемых на графах специального вида [122]. В этом случае станет возможным обучение непосредственно на чащах разбора, без использования проекций.

4. Поиск тождественных денотатов в онтологиях и формальных контекстах

4.1 Введение

Одним из типов связей, использовавшихся в предыдущих главах для соединения фрагментов текста, является анафора. Это частный случай отношения, которое на языке описания онтологий называется «та же сущность», а в лингвистике – кореферентность. Такая связь возникает при наличии нескольких наименований, ссылающихся на один и тот же объект или ситуацию «внеязыкового» мира. Обнаружение кореферентных связей (причем не обязательно парных) в тексте произвольного объема является отдельной задачей, известной в философии ещё с 19-го века под названием *задачи выявления тождественных денотатов* [5,6,7,8]. В общем случае эта проблема весьма сложна и требует построения сложных семантических моделей, а также использования дополнительных баз знаний. Однако в частном случае, когда мы имеем дело с формальными описаниями, построенными с помощью предварительной обработки текстовых данных, оказывается возможным предложить достаточно эффективные методы решения этой задачи [2, 15, 39].

Одной из наиболее универсальных и популярных моделей представления структурированных данных являются прикладные онтологии. Распространенным способом построения прикладной онтологии является её автоматическая или полуавтоматическая генерация из неструктурированных данных (как правило, текстов) на основе заранее подготовленного набора правил [139]. Однако при таком способе построения онтологии возникает проблема появления нескольких описаний, обозначений (денотатов) одних и тех же объектов реального мира. Возникновение данной проблемы в

рассматриваемом приложении связано с тем, что реальные источники информации могут существенно дублировать или перекрывать друг друга: например, во многих статьях может описываться одна и та же компания, человек, место и т.д.

При этом выявление тождественных денотатов непосредственно на этапе построения или дополнения онтологии (например, путем попарного сравнения новых объектов с уже существующими объектами) является не слишком эффективным сразу по нескольким причинам. Во-первых, такой подход существенно увеличивает нагрузку на эксперта, принимающего окончательное решение, в особенности эта нагрузка возрастает при частом обновлении данных. Во-вторых, в реальности тождественные объекты поступают неравномерно, и имеет смысл выявлять их не при каждом обновлении онтологии, а через более продолжительные промежутки времени, определяемые особенностями предметной области.

Предлагаемый подход позволяет эффективно выявлять тождественные денотаты в исходных данных, представленных в виде онтологии. Разработанный метод может либо автоматически формировать списки тождественных объектов, либо работать в качестве рекомендательной системы для эксперта, одновременно минимизируя нагрузку на него и предоставляя ему четкие и интуитивно понятные рекомендации по определению тождественных описаний объектов.

Задача, послужившая толчком к проведению исследования, была поставлена аналитиками компании Авикомп. Основное направление – поиск тождественных описаний людей и компаний в онтологиях, строящихся путем автоматической семантической обработки потока новостных текстов. Изначально задача решалась

методами попарного сравнения на основе расстояния Хэмминга и различными дополнительными эвристиками, причем качество решения было неудовлетворительным из-за низкой точности. Применение нового подхода позволило улучшить качество решения.

Прикладные онтологии, описывающие различные предметные области, в особенности, социальные сети, имеют специфические свойства, которые учитывались при разработке алгоритма:

1. Онтологии содержат достаточно большое количество объектов (десятки тысяч). Многие объекты имеют редкие или даже уникальные значения признаков, поэтому в онтологии содержится большое количество различных значений признаков.
2. Объекты содержат различное число выявленных признаков и связей (горизонтальных отношений) с другими объектами. Распределение этих чисел не линейное, а имеет «гиперболическую» форму (распределение Ципфа).
3. Другой особенностью задачи является «неравносильность» ошибок первого и второго рода. Ошибка первого рода (принятие двух описаний одного объекта за разные объекты) приводит к тому, что объекты онтологии содержат неполную информацию об объектах реального мира. Ошибка второго рода (объявление двух различных объектов тождественными) приводит к более серьезным последствиям – а именно, к введению в онтологию неверной информации об объекте.

4.2 Алгоритм поиска тождественных денотатов

Ниже описан разработанный алгоритм поиска тождественных денотатов в прикладной онтологии, который основан на методах анализа формальных понятий.

На вход алгоритм принимает прикладную онтологию. Онтология содержит объекты разных классов, объекты могут быть связаны отношениями, соответствующими их классам. Количество выявленных признаков и связей объекта может сильно варьироваться. Некоторые объекты описывают один и тот же объект реального мира.

На выходе алгоритм выдает списки объектов, которые были идентифицированы им как тождественные. Выявление объектов в онтологии осуществляется на основе объединения замкнутых множеств объектов с помощью методов анализа формальных понятий [26].

Алгоритм состоит из двух этапов. Первый этап - преобразование онтологии в формальный контекст. Второй этап - построение множества формальных понятий контекста онтологии и порождение списков тождественных объектов, производимое на основе отобранных по специальному критерию формальных понятий. Отметим, что второй этап может рассматриваться как самостоятельный алгоритм поиска тождественных денотатов в формальном контексте.

При этом алгоритм должен обладать высокой точностью, так как объявление двух различных объектов тождественными считается более грубой ошибкой, чем не обнаружение денотатов какого-либо объекта.

4.2.1 Преобразование онтологии в формальный контекст

Сначала исходные данные, представленные в виде (экземпляра) онтологии, преобразуются в так называемый *многозначный контекст*, задаваемый следующим образом:

1. Множество **объектов контекста** - это множество O объектов исходной онтологии.
2. Множество **признаков контекста** - это множество $M = L \cup C \cup R$, где:
   - L - множество атрибутов исходной онтологии,
   - C - множество бинарных признаков, совпадающее с множеством классов из структуры онтологии,
   - R - множество бинарных признаков, описывающих связи между объектами онтологии. Каждая связь $(x, y) \in instr(P)(p \in P)$ в онтологии порождает два бинарных признака в контексте: $p(x, \_)$ и $p(\_, y)$. Они соответствуют связи p, идущей от объекта x, и связи p, идущей к объекту y. Таким образом, объект x будет обладать признаком $p(\_, y)$, объект y – признаком $p(x, \_)$.
3. Каждый объект g получает следующие **значения атрибутов**:
   - Для признака из исходной онтологии $l \in L$:

$$l(g) = \begin{cases} instl(L), \text{если } attr(l) = inst(g) \\ null \text{ в противном случае} \end{cases}$$

   - Для признака $c \in C$:

$$c(g) = \begin{cases} True, \text{если } (inst(g), c) \in \left(H^C\right)^* \\ False \text{ в противном случае} \end{cases},$$

где $\left(H^C\right)^*$ - транзитивное рефлексивное замыкание отношения H^C.

- Для атрибута $r \in R$ вида $p(x, \_)$:

$$r(g) = \begin{cases} True, \text{если } (x, g) \in instr(p) \\ False \text{ в противном случае} \end{cases}$$

- Для атрибута $r \in R$ вида $p(\_, x)$:

$$r(g) = \begin{cases} True, \text{если } (g, y) \in instr(p) \\ False \text{ в противном случае} \end{cases}$$

Иными словами, каждый объект получает:

- Значения своих исходных атрибутов;
- Специальное значение *null* для атрибута *a*, если:
    - значение данного атрибута для него неизвестно;
    - атрибут *a* у данного объекта отсутствует.
- Бинарные признаки, соответствующие классу объекта и каждому его надклассу;
- Бинарные признаки, соответствующие его связям с другими объектами.

Данный подход к преобразованию позволяет учесть всю информацию об объекте, содержащуюся в исходной онтологии.

После получения многозначного контекста из онтологии нам необходимо построить бинарный (формальный) контекст. Для этого каждый признак многозначного контекста преобразовывается в несколько бинарных признаков. Данный процесс называется *шкалированием* [28]. Признаки многозначного контекста из множеств *C* и *R* изначально имеют бинарный вид, поэтому в преобразовании не участвуют. Признаки из множества *L* шкалируются в зависимости от типа признака. Как правило, большая часть признаков описывает неколичественные свойства объекта (например, имя человека, название компании и т.д.). К тому же многие из количественных или

просто числовых признаков таковы, что приближенное сходство по этим признакам не говорит о сходстве объектов. К примеру, если два объекта-компании имеют значения признака «Год создания» 2005 и 2006, то близость (но не совпадение) значений этого признака не повышает уверенность в том, что объекты описывают одну и ту же компанию, а скорее дает обратный эффект. Для таких признаков имеет смысл только совпадение значений признака. Если значения различны, то расстояние между ними не имеет значения. Такие признаки шкалируются *номинальной шкалой*, то есть каждому значению признака соответствует свой бинарный признак. К остальным количественным признакам могут применяться другие типы шкалирования, такие как:

- *Интервальное*: преобразование признака A во множество бинарных признаков вида «$a \le A < b$». При этом интервалы $[a,b)$ могут быть как непересекающимися, так и с перекрытием.
- *Порядковое*: признак A преобразовывается во множество бинарных признаков вида «$A > b$».
- Другие виды шкалирования, которые, по мнению эксперта, могут лучшим образом характеризовать сходство объектов как дублей.

В описанных ниже экспериментах на сгенерированных данных и реальной онтологии использовалось только номинальное шкалирование, однако это не ограничивает общности предложенного подхода.

4.2.2 Построение множества формальных понятий

По полученному формальному контексту строится множество формальных понятий. Существует несколько эффективных методов

нахождения формальных понятий. В нашем исследовании использовался алгоритм AddIntent [29].

Время работы данного алгоритма асимптотически равно $O(|L|*|G|^2*\max(|\{g'\}|, g \in G))$, где $|L|$ - количество формальных понятий контекста, G - множество объектов контекста, $|\{g'\}|$ - число признаков, которыми обладает объект.

Алгоритм довольно эффективен для работы с контекстами, полученными из онтологий, так как такие контексты содержат относительно небольшое число формальных понятий и б*о*льшая часть объектов имеет всего несколько признаков.

Один из альтернативных подходов построения формальных понятий основан на построении надпонятий для уже найденных понятий. Этот подход реализован, например, в алгоритме Замыкай-по-Одному [3]. Его преимуществом является возможность остановки алгоритма при достижении определенного размера понятий. Это свойство позволяет порождать не все понятия контекста, а только понятия с небольшим объемом, так как большие группы объектов скорее всего не являются дубликатами одного и того же реального объекта.

4.2.3 Критерии фильтрации формальных понятий

После построения множества формальных понятий необходимо выделить те формальные понятия, объем которых содержит только тождественные объекты.

При подборе критериев были учтены основные свойства, которыми должны обладать эти понятия. Во-первых, **критерий должен принимать б*о*льшее значение, если, при прочих равных, число признаков, которыми отличаются объекты понятия, будет**

меньше. В качестве критерия, характеризующего "разброс" признаков, был использован следующий индекс:

$$I_1(A,B) = \frac{|A||B|}{\sum_{g \in A} \left| \{g\}' \right|}$$

Максимальное значение индекса ($I_1 = 1$) достигается в случае, если ни один из объектов понятия не обладает признаками, не входящими в содержание понятия. Значение индекса стремится к нулю при уменьшении содержания понятия и увеличения у объектов понятия числа признаков вне содержания понятия.

Второе свойство, которым должен обладать критерий - **увеличение значение индекса при увеличении числа общих признаков (при прочих равных)**. При этом необходимо учитывать частоту признака. Распространенный признак должен делать меньший вклад в значение критерия, чем редкий, так как чем признак более распространен, тем больше шансов, что понятие с данным признаком возникло из-за случайного пересечения признаков.

В результате был разработан индекс, обладающий этим свойством:

$$I_2(A,B) = \sum_{m \in B} \frac{|A|}{\left| \{m\}' \right|}$$

Легко заметить, что появление нового признака в содержании формального понятия (при прочих равных) увеличивает значение индекса. При этом чем больше объектов в контексте обладают этим признаком, тем меньше изменится значение индекса.

Итоговый критерий *DII* представляет собой комбинацию описанных выше индексов. В данной работе использовались следующие способы комбинации:

1. Линейная комбинация описанных индексов: $DII_{+} = k_1 I_1 + k_2 I_2$
2. Произведение индексов со степенными коэффициентами: $DII_{*} = I_1^{k_1} * I_2^{k_2}$

Так как абсолютные значения коэффициентов влияют только на значение порога, а качество фильтрации будет определяться соотношением коэффициентов в формуле критерия, можно сузить семейство критериев без потери оптимального, взяв 1 в качестве значения одного из коэффициентов. Тогда семейства критериев будут представлены в виде формул с одной степенью свободы:

$$DII_{+} = I_1 + kI_2$$

$$DII_{*} = I_1 * I_2^{k}$$

Следует отметить, что при ранжировании с помощью произведения формальные понятия, для которых значения одного из индексов равны или близки к нулю, окажутся в конце списка. То есть этот способ делает обязательным наличие у понятия обоих свойств, описанных выше. Неизвестный коэффициент может определяться с помощью экспертной оценки, так как интерпретация обоих индексов, использованных в критерии, достаточно легка для понимания.

Другой подход к подбору коэффициента заключается в оптимизации критерия качества ранжирования формальных понятий. Алгоритм подбора коэффициента получает на вход множество формальных понятий одного контекста с разметкой: '1' - все объекты понятия - дубли, '0' - понятие содержит различные объекты. Далее рассматривается сетка на положительной вещественной оси, и на ней

максимизируется метрика качества ранжирования. Одна из таких метрик - Mean Average Precision (MAP) - более подробно описана ниже.

4.2.4 Формирование списков тождественных объектов

Списки объектов, которые алгоритм будет выдавать в качестве тождественных, формируются на основе объемов формальных понятий с высоким значением критерия. Алгоритм предусматривает два режима работы: автоматическое принятие решения и полуавтоматический режим с привлечением эксперта-аналитика.

В автоматическом режиме подразумевается, что аналитик не участвует в принятии решения о том, являются ли объекты формального контекста тождественными. Формирование состоит из двух этапов.

Первый этап заключается в фильтрации формальных понятий по порогу на разработанный критерий. На этом шаге могут добавляться различные эвристики, которые трудно учесть с помощью критерия. В результате фильтрации формируется список формальных понятий с высоким значением критерия.

Второй этап заключается в формировании списков тождественных объектов. Поскольку предполагается, что отношение «быть тождественным» (здесь и далее - на множестве объектов) транзитивное, а объекты отобранного формального понятия связаны этим отношением друг с другом, задача формирования списков таких объектов сводится к поиску компонент связности этого отношения на множестве объектов.

Построение списков осуществляется следующим образом. Вначале строится симметричное отношение R «быть тождественным».

Для каждого формального понятия с объемом $\{g_1,\ldots,g_n\}$ в отношение R добавляются пары $(g_1,g_i), g \in 2\ldots n$. Затем по построенному отношению находятся компоненты связности алгоритмом на основе обхода в ширину. Полученные компоненты связности будут соответствовать спискам объектов, выделенных как тождественные друг другу.

Альтернативный режим работы алгоритма подразумевает, что решение о том, являются ли объекты понятия тождественными, принимает аналитик. В этом случае алгоритм предлагает понятия аналитику в порядке уменьшения «уверенности», что его объекты тождественны друг другу.

Алгоритм последовательно предлагает аналитику оценить понятия, упорядоченные по убыванию значений критерия *DII*. При этом списки формируются по мере получения ответов аналитика.

Перед тем как предлагать понятие аналитику для оценки, находятся все списки тождественных объектов, имеющие пересечение с объемом понятия. Если объем уже вкладывается в один из списков дублей, то понятие не предлагается аналитику для оценки, В противном случае, аналитику предлагается оценить понятие. Если аналитик дает положительный ответ по формальному понятию (считает, что объекты этого понятия тождественны друг другу), то алгоритм редактирует списки:

- Если объем понятия не имеет ни одного пересечения со списками, он добавляется как новый список.
- Если объем понятия имеет одно пересечение, то объекты из объема добавляются к списку, с которым есть пересечение.

- Если объем понятия имеет пересечения с несколькими списками, то эти списки объединяются в один и пополняются объектами из объема.

Таким образом, алгоритм получает список объектов, соответствующий текущей разметке аналитика. Соответственно, аналитик может на каждом шаге остановить процесс оценки понятий и получить сформированные списки.

Ранжирование формальных понятий по индексу *DII* позволяет давать эксперту в первую очередь понятия, которые с большей вероятностью содержат тождественные объекты, что значительно упрощает работу эксперту по сравнению с оценкой понятий в произвольном порядке.

4.3 Альтернативные методы

В ходе исследования был проведен сравнительный анализ разработанного метода с несколькими из наиболее распространенных методов, способных решать данную задачу. Использовавшиеся методы основаны на анализе формальных понятий и попарном сравнении объектов контекста. В качестве методов попарного сравнения объектов были рассмотрены методы на основе *расстояния Хэмминга* и *абсолютного сходства*. Данные методы могут быть применены к многозначному контексту или напрямую к онтологии, но для простоты сравнения они будут описаны в применении к бинарному контексту.

4.3.1 Метод на основе экстенсиональной устойчивости понятия

Устойчивость формального понятия была впервые введена в [4]. Позднее в работах [3,9] было предложено различать два типа устойчивости: экстенсиональную и интенсиональную. В данной

работе использовалась экстенсиональная устойчивость, так как предполагается, что тождественные объекты должны быть сильно связаны большим количеством признаков и иметь небольшое количество отдельных признаков, соответственно, формальное понятие, которое они образуют, должно быть устойчиво к удалению отдельных признаков.

Алгоритм поиска тождественных объектов аналогичен основному методу: из множества формальных понятий выделяются наиболее (экстенсионально) устойчивые понятия. Затем предполагается, что объекты из объема устойчивого формального понятия являются тождественными. По множеству выбранных формальных понятий строится отношение «быть тождественным» *R*. Затем находятся компоненты связности данного отношения. Полученные компоненты выдаются на вход в качестве итоговых списков объектов.

4.3.2 Метод на основе меры абсолютного сходства

Данный метод основан на попарном сравнении объектов. Предполагается, что объекты онтологии, являющиеся тождественными, имеют большое количество общих признаков. Поэтому в качестве критерия близости объектов используется количество их общих признаков. Индикатор, основанный на данной мере, представляет собой порог на количество общих признаков.

Алгоритм получает на вход квадратную матрицу близости $A: A[i][j] = k \Leftrightarrow$ i-й и j-й объекты имеют k общих бинарных признаков, а также порог $t(N)$.

По матрице A и порогу строится матрица смежности $B: A[i][j] > t \Rightarrow B[i][j] = 1$. Матрица смежности (аналогично входной

матрице) является симметричной и описывает некоторое отношение близости *R*. Исходя из того, что отношение «быть тождественным» является отношением эквивалентности и обладает свойством транзитивности, по полученному отношению *R* строится его транзитивное замыкание R^*. Классы эквивалентности в R^* соответствуют группам тождественных объектов. Тот же результат можно получить, выделив все компоненты связности отношения *R*.

Асимптотическая сложность алгоритма по времени - $O\left(n^2 * m\right)$, где *n* - количество объектов в формальном контексте, *m* - количество признаков.

4.3.3 Метод на основе расстояния Хэмминга

Алгоритм поиска дублей основан на попарном сравнении объектов. В качестве метрики близости используется расстояние Хэмминга. Вначале составляется квадратная матрица расстояний между объектами. Затем, по построенной матрице *A* и заданному порогу $t(N)$ строится матрица *B* отношения «быть тождественным» $R: A[i][j] > t \Rightarrow B[i][j] = 1, (x_i, x_j) \in R$. Полученное отношение симметрично и рефлексивно. По данному отношению находятся компоненты связности. Объекты, попавшие в одну компоненту связности, считаются тождественными.

Асимптотическая временная сложность алгоритма в худшем случае аналогична сложности алгоритма на основе абсолютного сходства - $O\left(n^2 * m\right)$, где *n* - количество объектов в формальном контексте, *m* - количество признаков.

4.4 Экспериментальные исследования

4.4.1 Эксперименты на формальных контекстах

4.4.1.1 Схема эксперимента

Для того чтобы получить статистические оценки качества разработанного алгоритма, основные эксперименты проводились на искусственно сгенерированных данных с заранее известными тождественными объектами. Это позволило оценить качество метода на большом количестве входных данных и провести количественное сравнение разработанного метода с наиболее распространенными альтернативными подходами. Наряду с этим, при генерации данных также учитывались особенности прикладной онтологии, что позволяет экстраполировать полученные результаты на реальные данные.

Для оценки качества метода использовались различные метрики качества на искусственно сгенерированных контекстах. При этом генерируемые формальные контексты обладали свойствами контекстов, получаемых из прикладных онтологий.

Во-первых, генерируемые контексты содержали большое количество объектов и признаков. Количество объектов измеряется десятками тысяч. При этом количество бинарных признаков сравнимо с количеством объектов, так как многие объекты содержат уникальные или редкие признаки. Каждый объект обладает относительно небольшим количеством признаков. Их число обычно не превышает нескольких десятков. Поэтому контекст сильно разрежен, и, несмотря на большой размер контекста, число формальных понятий в нем относительно небольшое.

Во-вторых, количество признаков у объектов достаточно сильно варьируется и, как правило, удовлетворяет закону Мандельброта. То есть количество признаков примерно обратно пропорционально рангу

объекта среди объектов, упорядоченных по количеству признаков у них.

Третье свойство, которое было учтено при генерации контекста, это неравномерное распределение частот признаков. Как правило, частота признака обратно пропорциональна его рангу в последовательности, упорядоченной по частоте появления признака у объектов контекста.

На первом шаге при создании искусственных данных генерировался список уникальных объектов заданной длины. После этого генерировался входной контекст, включающий в себя уже не только уникальные, но и тождественные друг другу объекты. Для каждого уникального объекта «тождественный» объект в контексте генерировался следующим образом: каждый признак исходного объекта с фиксированной вероятностью добавлялся во множество признаков нового объекта. Для некоторых исходных объектов подобным образом создавалось несколько объектов.

Для проведения сравнительного анализа использовалось несколько метрик качества метода: полнота, точность, среднее значение полноты алгоритма при 100% значении точности, MAP. В качестве основных метрик использовались **полнота** и **точность** алгоритма.

Для того чтобы корректно определить полноту и точность, рассмотрим задачу поиска тождественных денотатов как задачу удаления из множества объектов онтологии тождественных друг другу объектов. Тогда выделенную алгоритмом группу объектов будем интерпретировать как удаление из онтологии всех объектов группы за исключением одного. Таким образом, мы определяем полноту и точность алгоритма:

$$Precision = \frac{\left|D_{dub} \cap D_{del}\right|}{\left|D_{del}\right|}$$

$$Recall = \frac{\left|D_{dub} \cap D_{del}\right|}{\left|D_{dub}\right|}$$

Здесь D_{dub} - количество «дублей» (если есть n тождественных объектов онтологии, считается, что среди них есть $n-1$ «дубль»), D_{del} - количество удаляемых объектов (если алгоритм выдал группу из n объектов, считается, что мы удаляем $n-1$ объект; причем если среди них есть k различных по построению объектов, то считается, что $k-1$ объект мы удалили неправильно).

Так как качество характеризуется комбинацией этих показателей, а все сравниваемые алгоритмы имели дополнительные параметры (пороги), то рассматривались зависимости полноты алгоритма от точности, путем прогона алгоритмов с различными входными параметрами.

Также для оценки использовалась метрика качества ранжирования MAP (Mean Average Precision):

$$Map(K) = \frac{\sum_{i=1}^{|K|} AveP(K_i)}{|K|}$$

$$AveP(k) = \frac{\sum_{c \in C_k} (P(c))}{|C_k|},$$

где K - множество контекстов, C_k - множество релевантных формальных понятий контекста k, $P(c)$ - доля релевантных понятий среди всех понятий, имеющих ранг не ниже, чем у понятия c.

4.4.1.2 Результаты

Для оценки нового метода сначала были подобраны оптимальные коэффициенты для индекса. Коэффициент подбирался по одному из сгенерированных контекстов. Бралась сетка на положительной вещественной оси, и на ней максимизировался индекс MAP.

Таким образом, были получены коэффициенты для использовавшихся вариантов индекса *DII*:

$$DII_{+} = I_1 + 0.25 I_2$$

$$DII_{*} = I_1 * I_2^{0.18}$$

Алгоритм с данным индексом сравнивался с альтернативными методами. Для построения зависимости точности алгоритма от его полноты для каждого метода задавалось несколько десятков различных порогов, затем рассчитывались полнота и точность алгоритма при каждом пороге. Эти показатели рассчитывались для нескольких сгенерированных контекстов, далее определялось среднее значение полноты и точности для каждого порога. Полученные соотношения позволяют сравнить использовавшиеся алгоритмы (рисунки 4.1, 4.2).

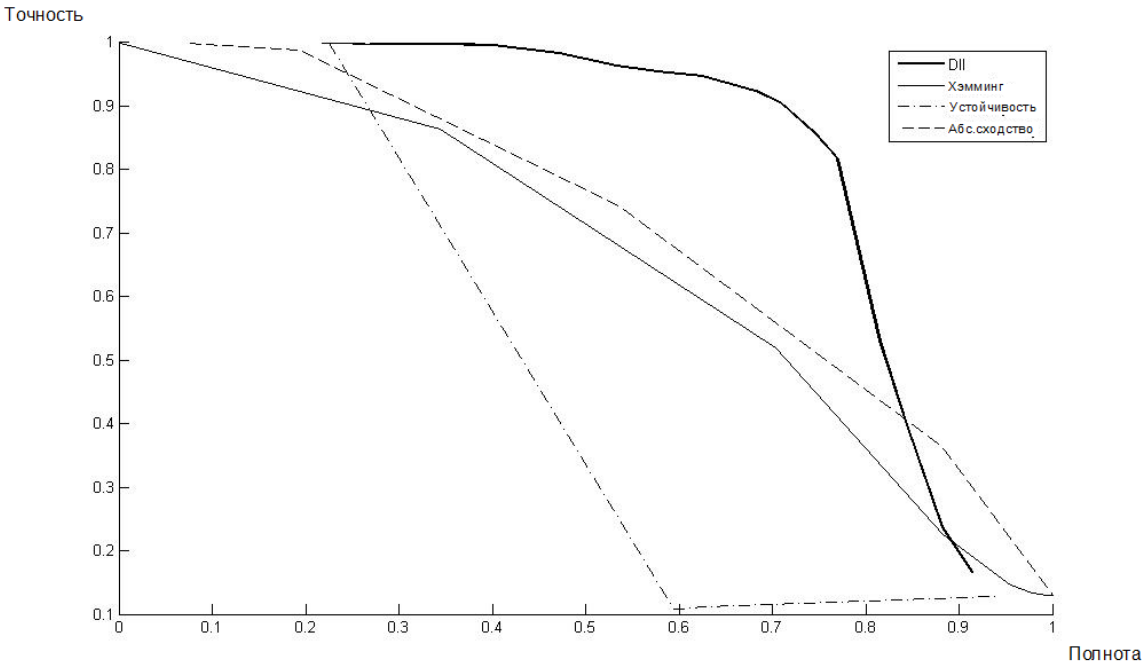

Рис. 4.1. Зависимость точности алгоритмов от полноты.

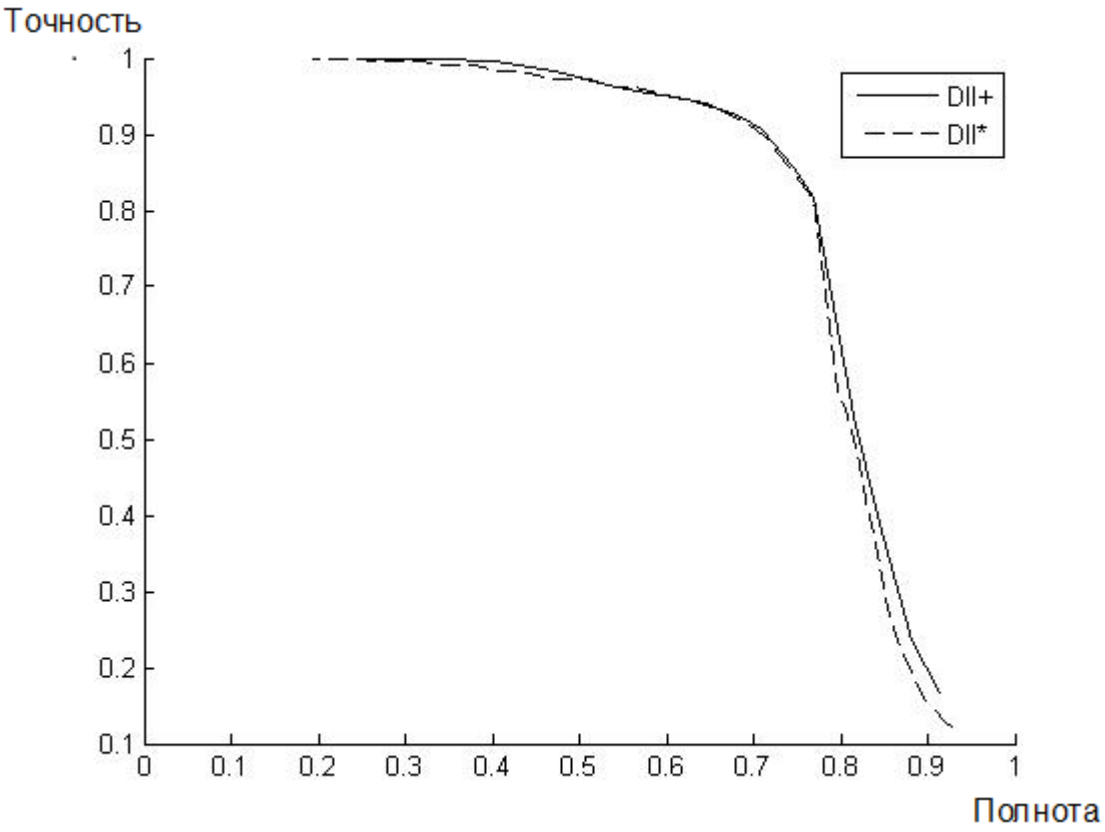

Рис. 4.2. Зависимость точности от полноты для двух вариантов нового индекса *DII*

Метод на основе экстенсиональной устойчивости показывает хорошие результаты при высоком пороге на индекс. При пороге больше 0.5 отбираются только формальные понятия, содержащие тождественные объекты. При пороге ниже 0.5 точность алгоритма падает в среднем до 10%, так как большое количество формальных понятий с устойчивостью 0.5 - однопризнаковые понятия, которые по определению не характеризуют тождественные объекты.

Алгоритм с использованием расстояния Хэмминга показал сравнительно низкие результаты. Так как расстояние может быть только целым положительным числом, снижение порога на 1 добавляет группу новых связей. При достаточно низком пороге точность близка к 100%, но даже среди объектов, имеющих одинаковый набор признаков, могут быть пары, не являющиеся тождественными. Как правило, это объекты с 1-2 распространенными признаками. Но расстояние Хэмминга не учитывает количество общих признаков, а только различия в признаках.

Алгоритм на основе абсолютного сходства объектов оказался наиболее эффективным среди рассмотренных альтернативных алгоритмов. В большинстве случаев большое количество общих признаков у пары объектов говорит о том, что объекты являются тождественными. Недостаток индекса в том, что он не учитывает различия объектов. К тому же некоторые признаки встречаются у большого количества объектов, и наличие их среди общих признаков не дает большого вклада в уверенность, что объекты являются тождественными.

Алгоритм на основе нового индекса (с использованием как одного, так и другого варианта комбинации) показал более высокие результаты, чем рассмотренные альтернативы. Основной

отличительной особенностью метода является небольшое падение точности алгоритма (до 90%) при росте полноты вплоть до 70%. По остальным метрикам данный метод показал высокие результаты. Результаты для DII_+ и DII_* оказались весьма схожими. Отличием DII_* стало менее стабильное поведение: иногда ошибаясь при большом пороге, в ряде случаев алгоритм не делал ошибок при малых порогах, выделяя при этом 42% тождественных объектов.

По показателю максимальной полноты без потери точности наиболее эффективным оказался метод на основе индекса устойчивости, который позволяет, поставив порог на индекс равным 0.5, выделять в среднем 22.44% тождественных объектов. При этом индекс DII_+ «отстал» по этому показателю незначительно, в отличие от методов попарного сравнения. Методы на основе попарного сравнения показали значительно более низкие результаты по данной метрике (таблица 4.1).

Таблица 4.1. Максимальная полнота алгоритмов при максимальной точности

Алгоритм	**Максимальная полнота при точности 100% (на экспериментальных данных)**
Алгоритм на основе абсолютного расстояния	6.22%
Алгоритм на основе расстояния Хэмминга	0.56%
Алгоритм на основе индекса устойчивости	22.44%
Алгоритм на основе нового индекса DII_+	21.78%
Алгоритм на основе нового	9.49%

индекса DII_*	

При сравнении методов на основе индекса экстенсиональной устойчивости и вариантов нового индекса DII_+ и DII_* по мере *MAP* очевидное преимущество имеет новый индекс (таблица 4.2).

Таблица 4.2. Результаты оценки по мере Mean Average Precision

Алгоритм	**MAP**
Алгоритм на основе индекса устойчивости	0.499
Алгоритм на основе нового индекса DII_+	0.935
Алгоритм на основе нового индекса DII_*	0.938

Для каждого метода был подобран оптимальный порог, при котором алгоритм имеет оптимальную полноту при минимальных потерях точности (таблица 4.3).

Таблица 4.3. Оптимальные пороги для методов и качество поиска

Алгоритм	Порог в алгоритме	Полнота	Точность
На основе абсолютного расстояния	3.50	19.35%	98.82%
На основе расстояния Хэмминга	0.50	34.37%	86.32%
На основе индекса устойчивости	0.50	22.44%	100%
На основе нового индекса DII_{+}	1.15	40.09%	99.58%
На основе нового индекса DII_{*}	0.90	31.80%	99.55%

4.4.2 Эксперименты на прикладной онтологии

4.4.2.1 Описание прикладной онтологии

Онтология, на которой был апробирован предложенный алгоритм, была построена компанией Авикомп. Онтология строилась и расширялась автоматически путем семантической обработки потока новостных сайтов программным средством OntosMiner [22].

По обработанному документу строится небольшая онтология с объектами и связями, выделенными в тексте. Затем онтология документа сливается с основной онтологией. Во время слияния происходит поиск тождественных объектов среди объектов основной онтологии и онтологии документа методом на основе расстояния Хэмминга с дополнительными эвристиками. При этом часто объекты, являющиеся тождественными, не идентифицируются как один объект, и в результате в онтологии возникает большое количество тождественных объектов, создающих избыточность в данных.

Анализируемая онтология была построена по новостным документам политической направленности. Она содержит 12006 объектов различных классов. Объекты имеют различное количество признаков и связей с другими объектами. Количество признаков и связей с другими объектами распределено по закону Ципфа.

В анализируемой онтологии был проведен поиск тождественных денотатов среди объектов классов «Персона» и «Компания». Таких объектов в онтологии 9821. Признаки формального контекста строились с использованием всех объектов и связей в онтологии.

4.4.2.2 Анализ результатов

Для получения точных оценок полноты и точности алгоритмов необходимо иметь информацию о том, какие объекты являются тождественными. Данную информацию можно получить лишь с помощью экспертной оценки коллекции обработанных документов. К сожалению, в силу специфики задачи (автоматическое построение онтологии и большой объем исходных документов), получить точную оценку полноты не представляется возможным.

Изначально алгоритм на основе индекса *DII* (использовался вариант DII_+) выделил около 900 групп объектов. В результате экспертной оценки было выявлено несколько ошибок. Алгоритм объединил объекты с разными именами/фамилиями, которые имели большое количество общих связей и признаков (партнеры, коллеги). Ошибка возникает из-за того, что алгоритм не учитывает, что различные значения некоторых конкретных признаков говорят о том, что объекты не являются тождественными. Поэтому в алгоритм было добавлено довольно простое дополнительное ограничение – отбрасывать понятия с объектами, у которых разные имена или фамилии. Стоит отметить, что подобное ограничение не распространяется на все признаки, так как они могут меняться со временем.

Далее метод использовался с дополнительными условиями. Алгоритм выделил 905 групп объектов. Размеры групп варьируются

от 2 до 41 объекта. Наиболее крупные группы, выделенные алгоритмом, описывают Нетаньяху Биньямина (41 объект), Юлию Тимошенко (35 объектов), Владимира Путина (34 объекта), Дмитрия Медведева (33 объекта), Стива Джобса (31 объект) и др. Но основная часть выделенных групп состоит из 2-3 объектов.

В результате оценки результатов работы алгоритма были получены оценки точности алгоритма. В 98% групп с высокой вероятностью можно утверждать, что объединенные в них объекты являются тождественными. Часто это следует из наличия у объектов таких общих признаков, как *фамилия* и *имя*. Также нередко встречаются группы, где данные признаки не являются общими, но по другим признакам и связям объекты объединяются в одну группу. Например, в онтологии было выявлено 7 объектов, описывающих Ксению Собчак. При этом часть объектов имели признаки «Фамилия:Собчак», «Имя:Ксения»", другая часть имели признаки «Имя:Ксения», «Отчество:Анатольевна». Несмотря на то что у объектов всего один общий признак (имя), за счет общих связей было выявлено, что это один и тот же объект. Аналогичная ситуация с объединением объекта с признаком «Имя:Усама» и объекта с признаком «Фамилия:Ладен».

Стоит также отметить, что наличие весов у признаков в индексе I_2 позволяет выделять большие группы объектов, описывающие Путина, Тимошенко, Медведева и т.д. Особенности данных объектов в тот, что каждый и них имеет большое количество собственных признаков, связей, поэтому расстояние Хэмминга между этими объектами довольно большое, а число общих признаков небольшое. Поэтому рассмотренные альтернативы, основанные на попарном сравнении объектов, плохо работают на данных объектах. При этом

формальное понятие, содержание которого состоит из имени и фамилии персоны, имеет высокое значение индекса *DII*, так как объекты понятия составляют значительную часть объектов, обладающих данными признаками. При этом его подпонятия имеют более низкое значение индекса *DII*.

4.5 Выводы

В данной главе был предложен алгоритм поиска тождественных объектов в прикладной онтологии (и формальном контексте), основанный на методах анализа формальных понятий. Метод состоит из двух основных этапов: преобразование онтологии в формальный контекст и формирование списков тождественных объектов с помощью отбора формальных понятий. Помимо метода решения задачи был разработан индекс, позволяющий ранжировать формальные понятия по степени уверенности в том, что объекты данного понятия тождественны.

Были рассмотрены альтернативные методы решения поставленной задачи, основанные на попарном сравнении объектов. Также был рассмотрен альтернативный критерий отбора формальных понятий, основанный на применении индекса экстенсиональной устойчивости.

Был произведен сравнительный анализ разработанного метода с его альтернативами и выявлены основные свойства всех методов. Сравнение методов производилось на случайно сгенерированных данных. При генерации были учтены все выявленные свойства реальной онтологии, что позволяет результаты, полученные на сгенерированных данных, перенести на реальные онтологии. Для сравнения были использованы основные метрики качества классификаторов (полнота, точность) и методов ранжирования (*MAP*).

Эксперименты на сгенерированных данных продемонстрировали преимущества нового метода. Эксперименты на реальных данных показали, что разработанные метод и критерий для фильтрации понятий довольно эффективны. На реальной онтологии алгоритм допустил всего несколько грубых ошибок, но при добавлении простейших дополнительных условий при отборе понятий алгоритм показал высокую точность. Экспертная оценка сформированных групп объектов не выявила явных ошибок.

5. Программные комплексы обработки текстовых данных на основе решеток замкнутых описаний

5.1 Программный комплекс FCART

5.1.1 Введение

Formal Concept Analysis Research Toolbox (*FCART*) – программный комплекс анализа данных методами АФП [26]. Этот продукт ориентирован на исследователей, пользующихся в основном методами Анализа Формальных Понятий, причём исходные данные для анализа уже представлены в виде, удобном для преобразования в объектно-признаковую форму.

Сейчас специалистам в области АФП известно несколько программных инструментов, таких как ConExp [11], Conexp-clj [24], Galicia [104], Tockit [25], ToscanaJ [105], FCAStone [106], Lattice Miner [107], OpenFCA [108]. Как правило, эти программы написаны на языке Java, являются кроссплатформенными и не требуют сложного развёртывания. Однако они не могут полностью удовлетворить запросы АФП-сообщества. Есть несколько областей, которые требуют серьёзных улучшений: средства подготовки данных (препроцессинга), расширяемость и масштабируемость, а также отсутствие универсальности и «отсталый» интерфейс с пользователем. Можно констатировать отсутствие универсальной интегрированной среды для поддержки разработки данных, выявления знаний и решения других задач методами АФП. Некоторые усилия в этом направлении приложили создатели *Tockit* [25], но затем основные усилия разработчиков переключились на создание *ToscanaJ*, которая стала специализированным продуктом. Таким образом, мотивацией для разработки FCART [40, 41] явилось создание универсального средства поддержки полного цикла исследований с использованием АФП.

5.1.2 Базовые понятия

5.1.2.1 Аналитические артефакты

Термин «аналитический артефакт» используется для обозначения абстрактного типа данных, описывающего некоторую сущность, возникающую в ходе анализа данных. Введение данного термина полезно, так как многие сущности встречаются многократно, могут быть формально описаны и типизированы. Например, в АФП фундаментальным артефактом является «формальный контекст», то есть объектно-признаковое представление части прикладной области. Другие важнейшие артефакты – «формальное понятие» и «решётка формальных понятий».

Артефакты связаны отношениями «являться источником данных для порождения». Например, из формального контекста можно получить решётку формальных понятий. В этом случае контекст является входом для алгоритма построения решётки. Другой пример – порождение ассоциативных правил на основе решётки.

В процессе анализа данных исследователь работает с экземплярами артефактов, то есть с конкретными данными. Любой экземпляр артефакта является «неизменяемым» [*immutable*]. Это значит, что пользователь не может изменить его после создания, хотя и может визуализировать в различных представлениях, включая интерактивные.

Имея предопределённый набор артефактов, поддерживаемых программой, мы можем использовать термины «тип» и «экземпляр» для различения абстрактного типа данных и конкретной порции этих данных. Но в большинстве случаев эти приставки к слову «артефакт» можно опускать без появления неоднозначности.

Коллекция всех экземпляров артефактов, накопленных в процессе исследования, называется «аналитической сессией». Артефакты, которые сгенерированы на основе внешних данных, считаются *базовыми*.

5.1.2.2 Решатели

Артефакты порождаются (или генерируются) *решателями (solvers)*. Каждый решатель представляет собой реализацию алгоритма построения одного набора артефактов на основе другого набора. Именно решатель фактически задаёт отношение «являться источником данных для» между артефактами.

Тип решателя – формальное описание его входов и выходов в виде двух последовательностей типов артефактов. Понятно, что программа может содержать несколько решателей одного типа, отличающихся используемыми алгоритмами, что может приводить к различиям в вычислительной сложности.

Использование методологии «решатель-артефакт» обусловлено вполне конкретными технологическими причинами. Имея предопределённый набор артефактов и решателей, программа может поддерживать целостность аналитической сессии. Без явного действия пользователя, программа не может удалить экземпляры артефактов. Напротив, для любого артефакта предусмотрена возможность перейти к его «предкам» или «потомкам»: здесь имеется в виду последовательность их генерации.

5.1.2.3 Визуализаторы

Визуализатор артефакта – это специальный вид решателя, который создаёт визуальное представление входного экземпляра артефакта заданного типа. С технической точки зрения, визуализатор строит интерактивное или неинтерактивное окно с некоторой

информацией и элементами управления (интерфейсом с пользователем). В простейшем случае визуализатор генерирует текстовое представление, отображаемое в стандартном текстовом редакторе (например, с подсветкой синтаксиса).

Очевидно, что для одного артефакта может существовать несколько визуализаторов. Также очевидно, что решателям можно не поддерживать интерфейс с пользователем (кроме запроса входных данных, который можно организовать стандартными средствами). Собственно, система запрещает решателям взаимодействовать с пользователем в процессе работы.

Обычно визуализатор запускается последним в цепочке из нескольких решателей. Но ничто не мешает пользователю в любой момент получить представление любого из артефактов в сессии.

В используемой методологии различается порождение нового артефакта и создание нового визуального образа для уже имеющегося артефакта. Это удобно по следующим причинам:

1. Явно разделяются решатели, способные запускаться в пакетном режиме для коллекции артефактов, и средства интерактивной работы пользователя.
2. Повышается эффективность обработки цепочек решателей.
3. Легко организуется исследование вычислительной сложности решателей.
4. Удачно реализуется разделение браузера сессии (показывающего все артефакты) и браузера окон (показывающего все активные визуализаторы и отчёты).

5.1.2.4 Отчёты

Отчёт – это итоговое представление результатов исследования. Полноценная программная среда проведения научных исследований обязана содержать редактор отчётов для предварительного оформления результатов. Основным отличительным признаком интегрированного редактора отчётов является возможность вставлять в отчёт выбранный элемент аналитической сессии, то есть некоторый артефакт. При этом вставка артефакта должна по возможности приводить к появлению как можно более полной информации в наиболее пригодном для дальнейшего редактирования виде: форматированном тексте, таблицах или векторной графике.

5.1.3 Программная архитектура комплекса

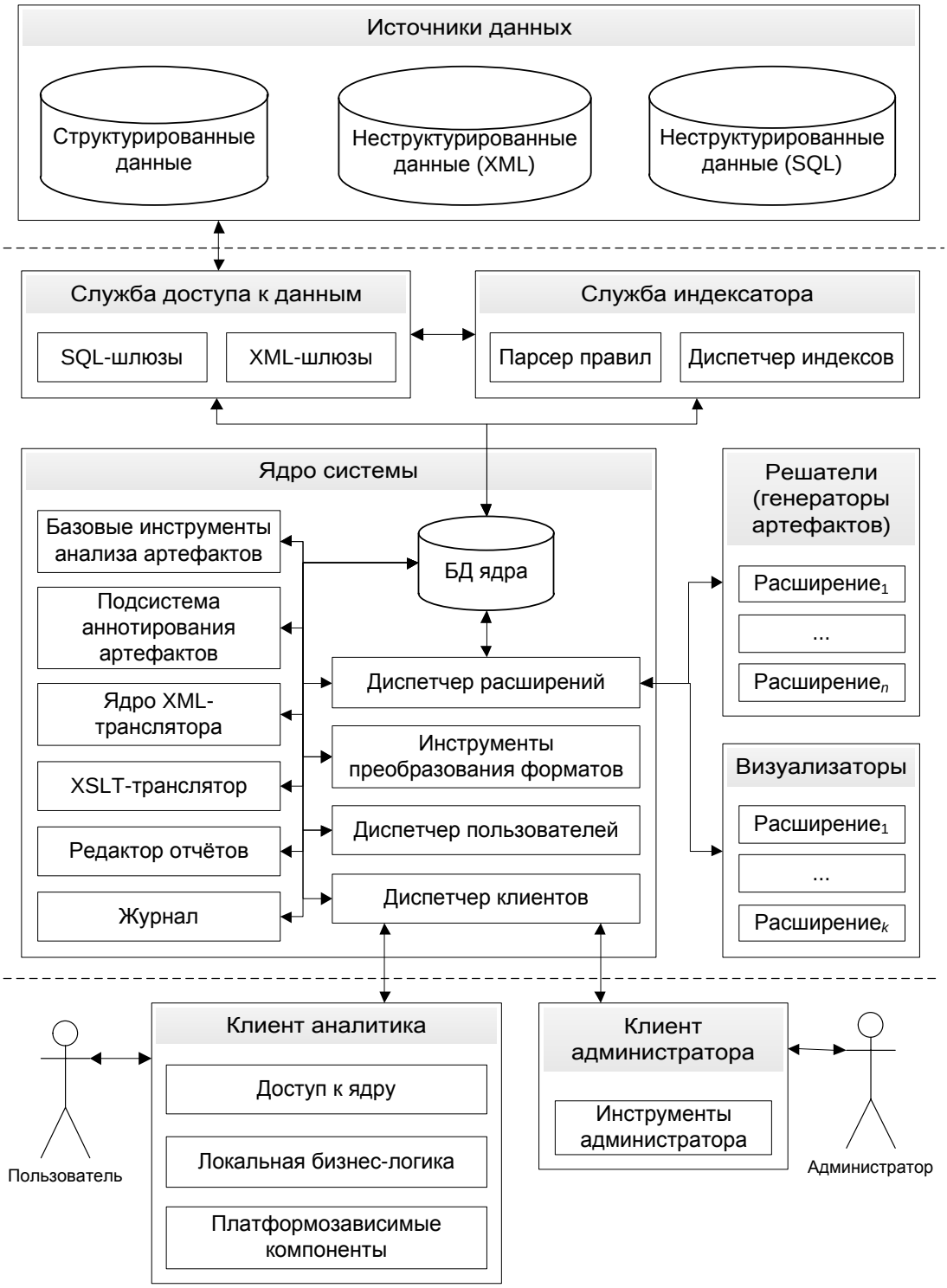

Рис. 5.1. Общая архитектура платформы *FCART*

С архитектурной точки зрения платформа представляет собой многокомпонентное распределённое приложение с возможностью расширения за счёт смены/дополнения компонентов доступа к

данным, решателей и визуализаторов. Рисунок 5.1 иллюстрирует архитектуру системы.

Таблица 5.1 даёт представление о спектре реально использованных при построении платформы технологий.

Таблица 5.1. Основные технологии, использованные при создании *FCART*

№	**Назначение**	**Технологии**	**Источники**
1	Базовые среды разработки	*Microsoft Visual Studio*, *Embarcadero RAD Studio*	www.microsoft.com/visualstudio, www.embarcadero.com/products/rad-studio
2	Базовые языки разработки	*C++*, *Delphi*, *C#*	msdn.microsoft.com/ru-ru/visualc/aa336395.aspx, msdn.microsoft.com/en-us/vstudio/hh388566.aspx, www.embarcadero.com/products/delphi
3	Хранилища данных	*BaseX*, *Sedna*	www.basex.org, www.sedna.org
4	Визуализация данных в *Web*-версии	*Microsoft SilverLight*	www.microsoft.com/silverlight
5	Создание макросов / скриптов	*Delphi Web Script*, *Python*	www.delphitools.info/dwscript, www.python.org
	Полнотекстовый поиск	*Apache Lucene*	lucene.apache.org

Базовые свойства системы:

1. Поддержка работы как в сети Интернет (*web*-версия), так и в локальном режиме (локальная версия);
2. Расширяемость функциональности без изменения базовых компонентов (поддержка плагинов [*plugins*] и макросов [*scripts*]);
3. Ориентация на визуальную интерактивную работу с артефактами анализа данных;
4. Встроенные редакторы отчётов, локальные БД, средства протоколирования и тестирования.

5.1.4 Цикл работы на примере решеток замкнутых описаний

Методология использования предусматривает итеративное построения базовых артефактов (контекстов) и изучение прочих артефактов с изменением запросов к данным при получении новых знаний об объекте исследования. Цикл анализа на примере артефактов АФП можно описать следующим образом:

1. Сначала создаются или открываются снимки данных (их может быть несколько, что требуется, например, при динамическом анализе данных). Снимок представляет собой таблицу, где строками являются объекты из *XML*-хранилища, а поля определяются профилем (*profile*). Для работы с профилями существует редактор, который может работать в режиме анализа произвольного *XML/JSON* описания объекта.
2. Затем на основе снимков по запросу генерируются формальные контексты. Запросы строятся при помощи оригинального языка, делящегося на четыре части:

2.1 простые правила обработки атомарных полей без шкалирования;

2.2 правила обработки атомарных полей с номинальными или порядковыми шкалами;

2.3 правила обработки многозначных (векторных) полей;

2.4 правила обработки неструктурированных текстов (используется индексатор). Оптимальным является генерация запроса путём выбора некоторых понятий из исходной структуры и задания дополнительных параметров. Редактор предоставляет весь спектр возможностей редактора графов.

Имея набор контекстов, аналитик автоматически получает решётки формальных понятий и возможность построения вторичных артефактов (например, множества ассоциативных правил или базиса импликаций). Визуализатор решёток (рисунок 5.2) предлагает развитые средства поиска объектов и признаков, сравнения понятий, построения подрешёток по заданным множествам объектов и признаков.

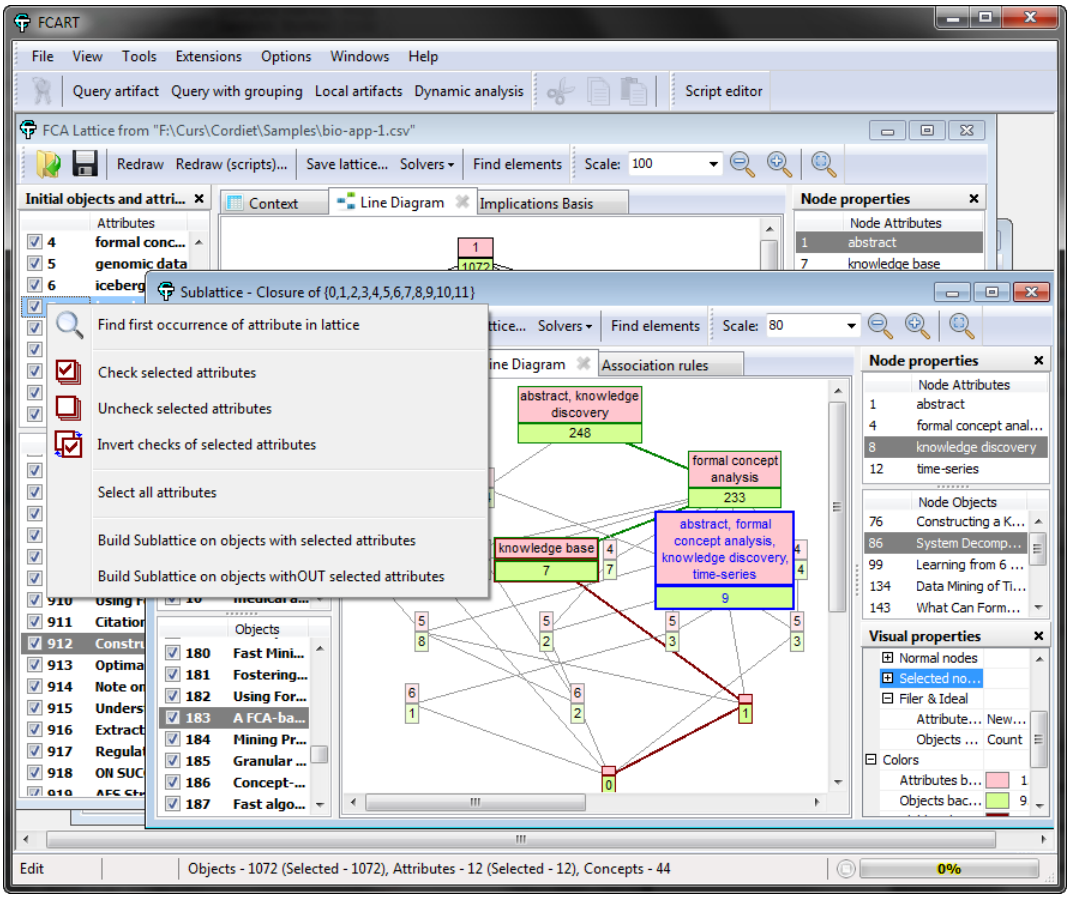

Рис. 5.2. Вариант визуализатора решётки формальных понятий

Масштабируемость системы позволяет в интерактивном режиме работать с решётками, содержащими более чем 10000 понятий (рисунок 5.3). При этом можно пользоваться дополнительными инструментами навигации по решётке, и управлением масштабом изображения, изменять внешний вид вершин (понятий) и отображаемую информацию.

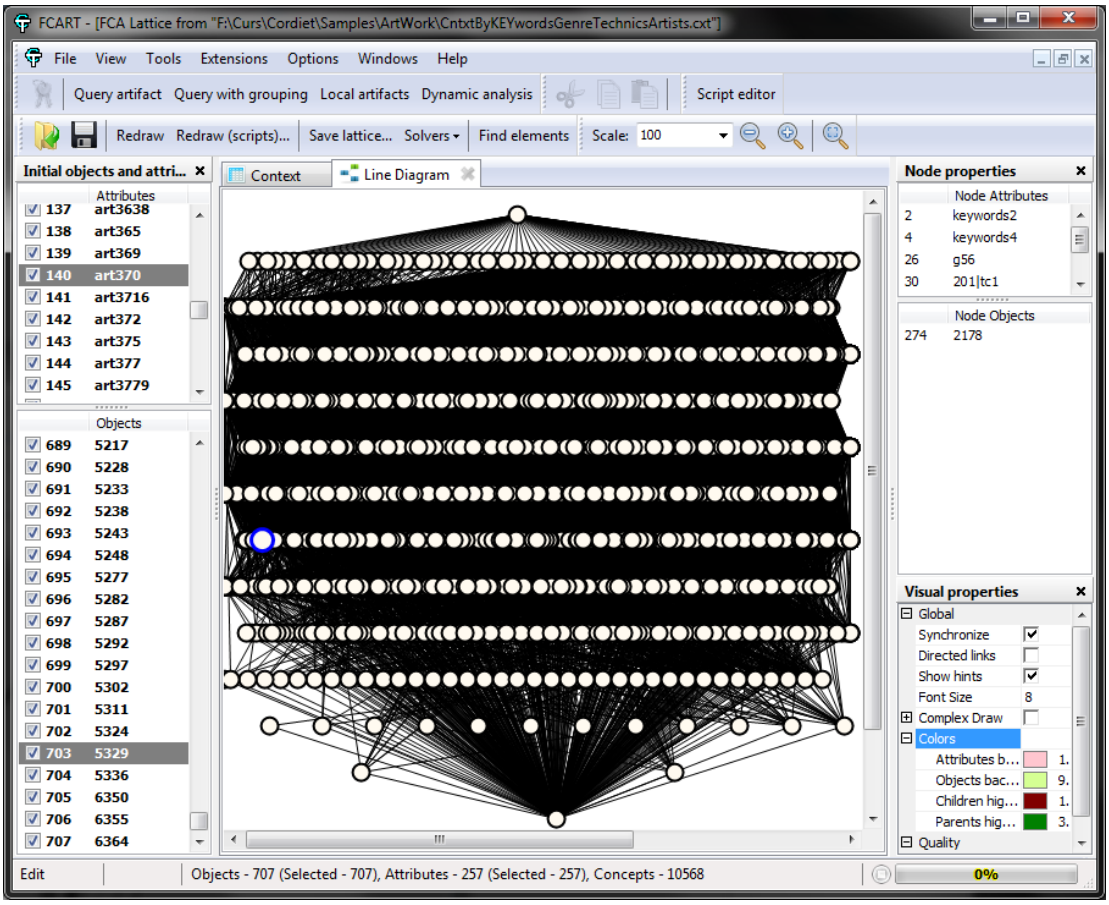

Рис.5.3. Проверка масштабируемости визуализатора решётки формальных понятий

Любой артефакт можно экспортировать в различных форматах. Обычно поддерживается как текстовое, так и графическое представление экспортируемых данных. Так, решётку можно сохранить в виде графа (в формате *XGMML*) или картинки (в векторном формате *EMF* или в растровых форматах *PNG*, *JPG*).

Встроенный редактор отчётов (рисунок 5.4) позволяет подготавливать полноценные отчёты с таблицами, иллюстрациями и ссылками на результаты вычислительных экспериментов.

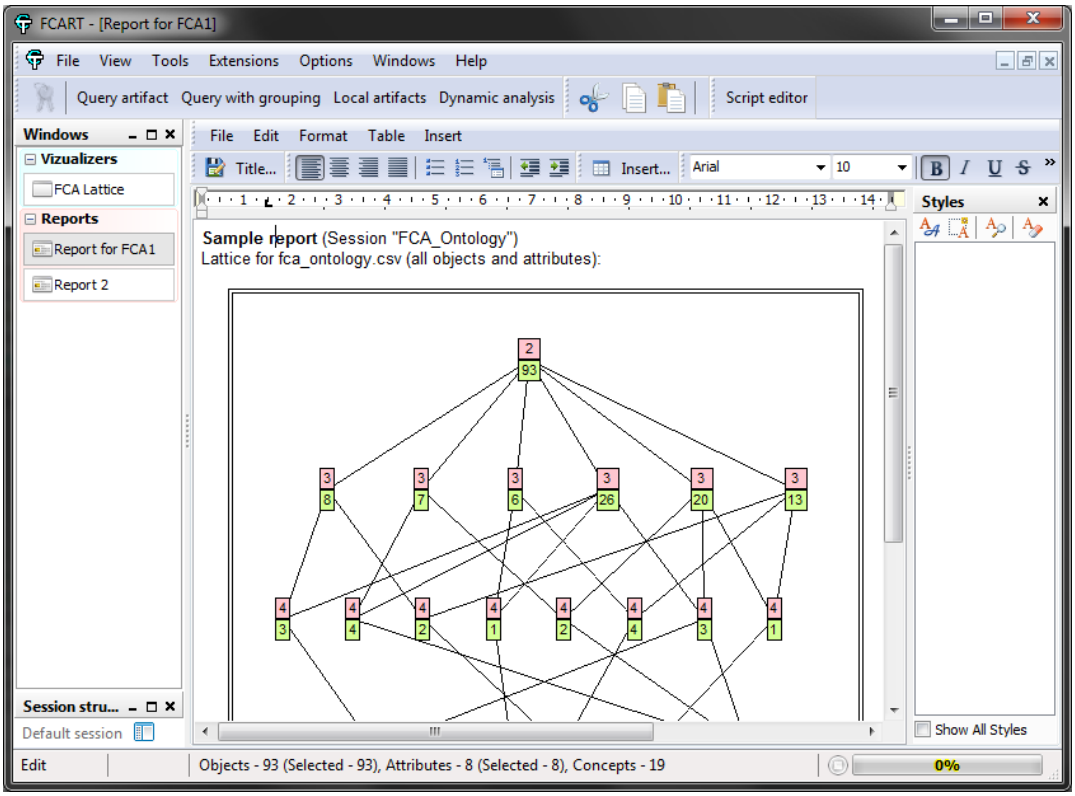

Рис.5.4. Редактор отчётов в FCART версии 0.7

5.1.5 Использование плагинов и макросов

Функциональность FCART расширяется за счет использованием так называемых плагинов и макросов. Отметим, что с использованием макросов можно существенно изменить поведение существующих решателей и визуализаторов или добавить новые, не перезапуская приложение. Встроенный редактор макросов/скриптов поддерживает подсветку синтаксиса и предоставляет полноценный контроль за ошибками, возникающими на этапе компиляции и во время выполнения программы (рисунок 5.5).

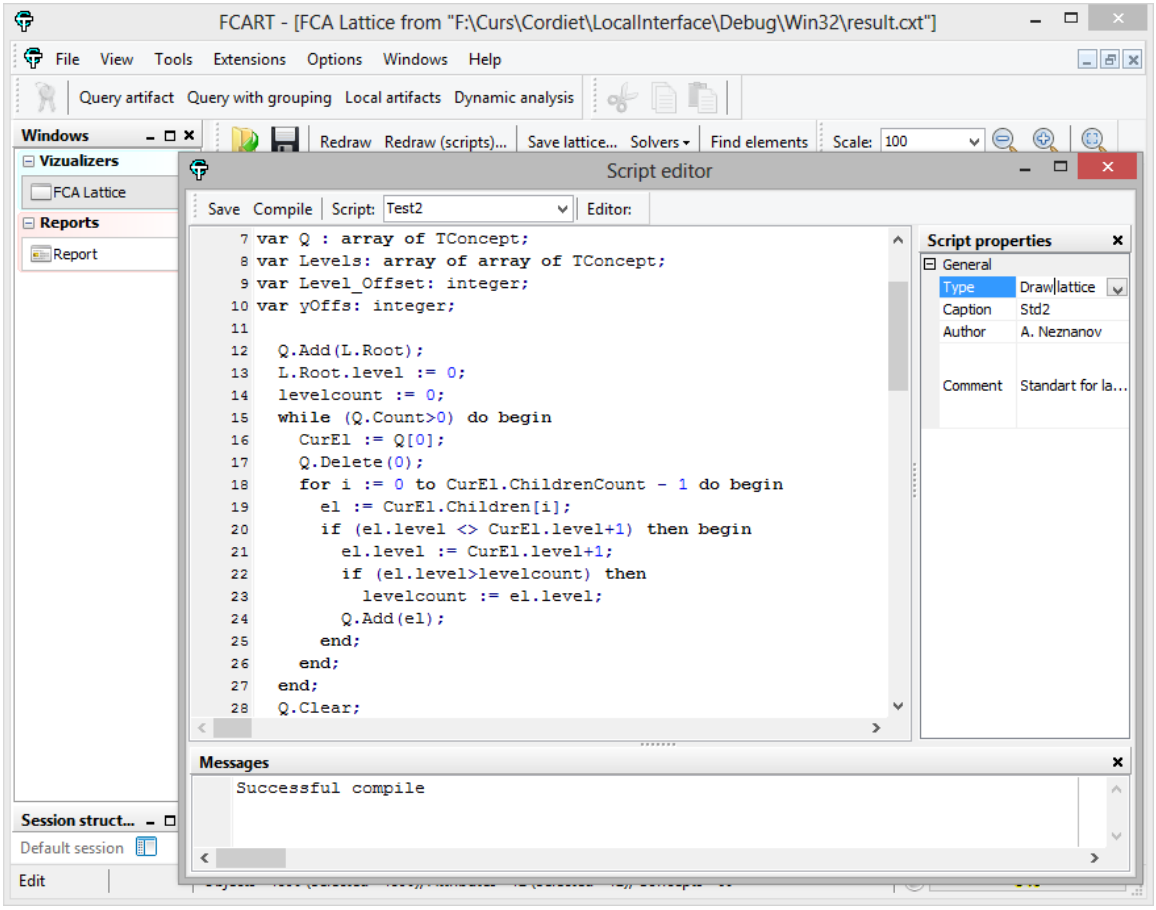

Рис. 5.5. Редактор макросов в FCART (редактируется алгоритм прорисовки диаграммы решётки формальных понятий)

5.1.6 Основные возможности программного комплекса по работе с решетками замкнутых описаний

На данный момент реализована локальная версия FCART. В этой версии присутствуют следующие основные возможности по работе с Анализом Формальных Понятий:

1. Предобработка данных:

1.1 Преобразование их в многозначный контекст с помощью специализированных запросов.

1.2 Шкалирование многозначных контекстов по заранее фиксированным шкалам, преобразование их в формальный контекст.

2. Работа с формальными контекстами с помощью визуальных решателей:

2.1 Загрузка, редактирование, сохранение формальных контекстов.

2.2 Автоматическое построение решеток понятий.

3. Работа с решетками формальных понятий с помощью визуальных решателей:

3.1 Построение порядковых фильтров и идеалов, айсберга решетки;

3.2 Преобразование и настройка визуального представления решетки;

3.3 Сохранение решетки в нескольких форматах.

4. Построение базисов ассоциативных правил и импликаций.

5. Фильтрация формальных понятий с помощью индексов (реализованы в виде скриптов):

5.1 Индексы экстенсиональной и интенсиональной устойчивости;

5.2 Индекс отделимости;

5.3 Разработанный в рамках данного диссертационного исследования индекс *DII*, который предназначен для выявления тождественных денотатов (см. главу 4).

6. Сохранение результатов в виде отчетов.

5.2 Программный комплекс, предназначенный для обработки чащ разбора

5.2.1 Архитектура комплекса

Данный программный комплекс предназначен для обработки текстовых данных. По классификации программных систем, приведенной в [14], он относится к системам специального назначения с открытой лицензией, которые могут быть использованы как часть других систем. Проект включает в себя следующие оригинальные модули:

- Модуль для работы с чащами разбора: построение, обобщение, вычисление проекций и т.д.;
- Модуль для построения узорных структур на чащах разбора и их проекциях;
- Модуль поиска: нахождение результатов и повторное ранжирование;
- Модуль обучения на абзацах: формирование обучающей и тестовой выборки, запуск процедуры обучения и т.д.
- Риторический парсер, основанный на правилах;
- Модуль для выделения коммуникативных действий.

В проекте используются следующие технологии и программные средства:

- OpenNLP/Stanford NLP парсеры [23,95] – для построения деревьев синтаксического разбора;
- Stanford NLP Coreference – для разрешения анафор и построения кореферентных связей;
- Bing API – для реализации базового поиска;

- Apache SOLR – для обеспечения интеграции с другими поисковыми системами;
- Риторический парсер Joty [124,125,126] – для автоматического построения дискурсивных деревьев на основе машинного обучения.
- TK-Light [113] – для обучения на деревьях с использованием ядер.

Код и библиотеки проекта доступны по ссылкам http://code.google.com/p/relevance-based-on-parse-trees и https://github.com/bgalitsky/relevance-based-on-parse-trees. Ключевые фрагменты кода приведены в Приложениях.

Архитектура комплекса предусматривает возможность интеграции с другими системами. В частности, он может быть подключен к библиотеке Lucene. Кроме того, в состав системы включен обработчик запросов SOLR, позволяющий интегрировать её в другие поисковые приложения, подключив к ним поиск по нескольким предложениям с использованием чащ для проверки роста релевантности.

5.2.2 Модуль обработки чащ разбора

Данный модуль предназначен для создания и обработки чащ разбора. Он включает в себя ряд функциональных возможностей:

1. Построение чащи из текстового абзаца;
2. Построение проекций чащи двух видов;
3. Нахождение сходства между чащами и между проекциями;
4. Экспорт чащи в виде графа;
5. Вывод чащи в виде текста.

5.2.3 Ранжирование поисковых результатов

Данный модуль обеспечивает переранжирование поисковых результатов с учетом сходства чащ разбора результата и запроса. Модуль позволяет вычислять значение релевантности для каждого обобщения чащ запроса и ответа, а также определять итоговый порядок с учетом этого значения.

5.2.4 Обучение на абзацах

Данный модуль интегрирован с процедурой обучения на ядрах для деревьев TK-Light. Он позволяет готовить обучающую и тестовую выборку, записывать результаты обучения в файлы.

5.2.5 Модуль кластеризации с помощью решеток замкнутых описаний

Модуль кластеризации представляет собой реализацию алгоритма AddIntent[29] для текстовых данных. На вход алгоритм принимает набор текстов. В качестве решеточной операции пересечения в алгоритме используется операция сходства на чащах (или на проекциях чащ, то есть на множествах расширенных групп).

5.2.6 Риторический парсер

Данный модуль предназначен для нахождения и обобщения расширенных групп, основанных на риторических отношениях. В экспериментах по поиску и поиску с помощью классификации для извлечения риторических связей использовался оригинальный риторический парсер, использующий правила. За основу при реализации парсера были взяты модели, описанные в работах [83,94]. При построении группы сначала находятся маркеры, свидетельствующие о наличии риторического отношения (как правило, это глаголы), затем устанавливается связь между двумя

синтаксическими глагольными группами в исследуемом тексте, выделяются риторические отношения.

В экспериментах по классификации технических документов использовался риторический парсер, разработанный Joty и др. [124,125,126]. Он основан на машинном обучении.

5.2.7 Модуль для выявления и обработки коммуникативных действий

Модуль позволяет выделять в тексте коммуникативные действия и их предикаты, устанавливать связи между коммуникативными действиями, а также выполнять обобщение получающихся расширенных групп. Для выявления коммуникативных действий и построения связей на их основе используется словарь коммуникативных действий, описанный в работе [73]. Для каждого термина из словаря используются 5 бинарных свойств.

5.2.8 Модуль для построения кореферентных связей

При автоматической обработке текстов на естественном языке важно правильно сопоставлять несколько раз упомянутые объекты. Для разрешения кореференций (coreference resolution) в программном комплексе используется модуль Coreference Resolution системы StanfordNLP [77, 123]. Он представляет собой набор детерминистических моделей, которые используют лексическую, синтаксическую и семантическую информацию, доступную на уровне всего документа.

Алгоритм, применяемый в системе, состоит из трех основных этапов:

1. Обнаружение упоминаний (сущностей).

2. Разрешение анафор.
3. Последующая обработка полученных данных.

На первом этапе извлекаются сущности вместе с информацией о них, такой как пол и число. На следующем этапе уже проходит непосредственное разрешение кореференций, последовательно применяется набор фильтров, начиная с наиболее точных. Пост-обработка позволяет, например, удалить упоминания, употребленные лишь единожды. На этапе извлечения сущностей используются ориентированные на полноту фильтры, в то время как для непосредственного разрешения кореференций уже нужна ориентация на точность.

На этапе разрешения кореференций используется следующий упорядоченный список фильтров:

1. Выявление упоминаний
2. Обработка семантической информации
3. Точное совпадение строк
4. Релаксированное совпадение строк
5. Совпадение структур
6. Совпадение начал
7. Совпадение имен собственных
8. Выявление псевдонимов
9. Релаксированное совпадение начал
10. Лексические цепочки (синонимия и гипонимия)
11. Фильтр местоимений

На этапе пост-обработки используются два фильтра: удаляются кластеры с одним элементом и отбрасываются упоминания, которые встречаются дальше в тексте в качестве аппозитива или соединения.

Заключение

В данной работе были рассмотрены различные модели представления абзацев текста: мешок слов, деревья синтаксического разбора, чащи синтаксического разбора. Также были рассмотрена теория решеток замкнутых описаний, введены понятия формального контекста, онтологии, решетки формальных понятий, узорной структуры и проекции узорной структуры. Помимо этого были кратко описаны теории дискурсивного представления абзацев текста, такие как теория риторических структур, теория речевых актов, теория дискурсивного представления текста и некоторые другие. Также было приведено описание методов обучения на структурах с использованием ядерных функций.

В работе была построена новая графовая модель текстов, использующая и обобщающая модель структурного синтактико-дискурсивного представления текстового абзаца (чащу разбора). Модель позволяет описывать сходство текстовых абзацев в терминах обобщения их структурных графовых и древесных описаний. В исследовании был предложен способ вычисления сходства между текстами, основанный на операции обобщения соответствующих им чащ разбора. В работе были реализованы точное и приближенное (с использованием проекций) обобщение чащ разбора. Было предложено несколько вариантов построения проекций представления и сходства структурных описаний. Было продемонстрировано, что применение проекций позволяет уменьшить временную и вычислительную сложность нахождения сходства между текстами, причем потеря информации является незначительной.

Модель была применена в задаче повторного ранжирования результатов информационного поиска по сложным запросам. Был

разработан численный метод повторного ранжирования, использующий предложенную модель. На нескольких наборах реальных интернет-данных из нескольких областей, предоставленных поисковым механизмом Bing, было продемонстрировано, что вычисление обобщения на уровне абзацев текста (обобщение чащ разбора) позволяет улучшить релевантность поиска по сравнению с деревьями разбора и мешком слов.

Было показано, что использование модели с введенной операцией обобщения позволяет построить таксономическое представление коллекции текстовых документов и применить представление в задаче иерархической кластеризации коротких текстов, повысив качество кластеризации. Кластеризация выполняется путем построения решетки замкнутых структурных описаний текстов.

В работе также было продемонстрировано, что предложенная модель применима к задаче классификации коротких текстов. На основе модели был разработан численный метод, использующий ядерные функции, определенные на деревьях. Данный метод был апробирован на задаче поиска с помощью классификации и на задаче классификации технических текстов. Было проведено сравнение двух вариантов обучения на текстах:

- Обучение на деревьях разбора для отдельных предложений (существующая модель текста),
- Обучение на деревьях разбора для отдельных предложений, дополненных расширенными деревьями разбора – деревьями, полученными на основе дискурсивных связей между предложениями абзаца (предложенная в исследовании модель текста).

Эксперименты продемонстрировали, что добавление новых признаков без изменения схемы эксперимента улучшает качество классификации с использованием существующей модели и устраняет недостатки, связанные с применением этой модели.

Также в работе были предложены новая модель и метод поиска тождественных денотатов в прикладной онтологии (и формальном контексте), основанные на применении анализа формальных понятий. Метод был применен для построения отношения «та же сущность», используемого в рассматриваемой в исследовании модели представления текстовых данных. Был разработан индекс, позволяющий ранжировать формальные понятия по степени уверенности в том, что объекты данного понятия тождественны друг другу. Были рассмотрены альтернативные методы решения поставленной задачи, основанные на попарном сравнении объектов, и альтернативный критерий отбора формальных понятий, основанный на применении индекса экстенсиональной устойчивости. Был произведен сравнительный анализ разработанного метода с его альтернативами и выявлены основные свойства всех методов. Эксперименты на сгенерированных данных продемонстрировали преимущества нового метода. Эксперименты на реальных данных показали, что разработанные метод и критерий для фильтрации понятий демонстрируют высокую точность.

В работе было также приведено описание программного комплекса FCART, в который в рамках исследования был добавлен индекс для вычисления тождественных денотатов, и программного комплекса для работы с текстовыми данными, объединяющего в себе реализацию предложенных в работе численных методов и алгоритмов.

Литература

1. Биркгоф Г. Теория решеток. — М.: Наука, 1989.
2. Ильвовский Д., Климушкин М. Выявление дубликатов объектов в прикладных онтологиях с помощью методов анализа формальных понятий. НТИ, Сер. 2. – 2013. - № 1. - С.10-17.
3. Кузнецов С.О. Быстрый алгоритм построения всех пересечений объектов из конечной полурешетки. НТИ, Сер. 2. – 1993. - № 1. - С.17-20.
4. Кузнецов С.О. Устойчивость как оценка обоснованности гипотез, получаемых на основе операционального сходства. НТИ. Сер.2 - 1990. - № 12. - С.21-29.
5. Карнап Р. Значение и необходимость. М., 1959.
6. Монтегю Р. Прагматика и интенсиональная логика. – В кн.: Семантика модальных и интенсиональных логик. М., 1981.
7. Фреге Г. Смысл и значение. – В кн.: Фреге. Избр. работы. М., 1997.
8. Рассел Б. Исследование значения и истины. М., 1999.
9. Теньер, Л. Основы структурного синтаксиса. / Пер. с франц. М.: Прогресс, 1988.— 656 с.
10. Мельчук, И.А. Опыт теории лингвистических моделей «Смысл ⇔ Текст». М., 1974 (2-е изд., 1999).
11. Евтушенко С.А. Система анализа данных «Concept Explorer». Труды 7-ой Национальной Конференции по Искусственному Интеллекту (КИИ-2000). – Москва. - 2000, С.127-134.

12. Климушкин, М., Четвериков, Д. Исследование американских политических блогов на основе анализа формальных понятий. ЗОНТ-09. - Новосибирск, РИЦ прайс-курьер. – 2009.

13. Ильвовский Д. Применение семантически связанных деревьев синтаксического разбора в задаче поиска ответов на вопросы, состоящие из нескольких предложений. НТИ. Сер.2 - 2014. - № 2. - С.28-37.

14. Ильвовский Д. А., Черняк Е. Л. Системы автоматической обработки текстов. Открытые системы. СУБД. 2014. № 01. С. 51-53.

15. Ильвовский Д. А., Климушкин М. А. Выявление дубликатов объектов в прикладных онтологиях на основе методов анализа формальных понятий. В кн.: Сборник докладов 9-й международной конференции ИОИ-2012. Торус Пресс, 2012. С.625-628.

16. Галицкий Б.А., Ильвовский Д.А. Выявление искаженной информации: подход с использованием дискурсивных связей // XV национальная конференция по искусственному интеллекту с международным участием КИИ-2016. Труды конференции. В 3-х томах. Смоленск: Универсум, 2016, 2, с.23-33.

17. Интернет-энциклопедия Википедия [Интернет-портал]. URL: https://en.wikipedia.org/wiki/Parse_tree (дата обращения: 06.07.2014).

18. Интернет-энциклопедия Википедия [Интернет-портал]. URL: https://en.wikipedia.org/wiki/Bag-of-words_model (дата обращения: 07.07.2014).

19. Интернет-энциклопедия Википедия [Интернет-портал] en.wikipedia.org/wiki/Cascading (дата обращения: 07.07.2014).
20. Ананьева М.И., Кобозева М.В. Дискурсивный анализ в задачах обработки естественного языка // Конференция «Информатика, управление и системный анализ», ИУСА, 2016.
21. Литвиненко А. О. Описание структуры дискурса в рамках Теории Риторической Структуры: применение на русском материале //Труды Международного семинара Диалог. – 2001. – С. 159-168.
22. Ontos [сайт]. URL - http://www.ontos.com/?page_id=630 (дата обращения: 07.07.2014).
23. The Stanford Natural Language Processing Group [сайт]. URL: http://nlp.stanford.edu/ (дата обращения: 10.07.2014).
24. Conexp-clj [сайт]. URL: http://daniel.kxpq.de/math/conexp-clj/ (дата обращения: 10.07.2014).
25. Tockit: Framework for Conceptual Knowledge Processing [сайт]. URL: http://www.tockit.org (дата обращения: 10.07.2014).
26. Ganter B., Wille R. Formal Concept Analysis: Mathematical Foundations. - Berlin: Springer, 1999.
27. Maedche, A., Zacharias, V. Clustering Ontology-based Metadata in the Semantic Web. Proc. of 6th European Conference on Principles of Data Mining and Knowledge Discovery. - 2002. - P. 348 – 360.
28. Prediger, S. Logical scaling in formal concept analysis. ICCS, Lecture Notes in Computer Science. – 1997. - Vol. 1257. Springer. - P. 332-341.

29. Merwe, D., Obiedkov, S., Kourie, D. AddIntent: a new incremental algorithm for constructing concept lattices. - LNCS, Springer. – 2004. – P. 205 – 206.

30. Kuznetsov, S.O., Obiedkov, S., Roth, C. Reducing the Representation Complexity of Lattice-Based Taxonomies. U. Priss, S. Polovina, R. Hill, Eds., Proc. 15th International Conference on Conceptual Structures (ICCS 2007), Lecture Notes in Artificial Intelligence (Springer), Vol. 4604, pp. 241-254, 2007.

31. Roth, C., Obiedkov, S., Kourie, D. On Succinct Representation of Knowledge Community Taxonomies with Formal Concept Analysis. IJFCS (Intl Journal of Foundations of Computer Science). – 2008. – P. 383-404.

32. Galitsky, B., Ilvovsky, D., Lebedeva, N., Usikov, D. Improving Trust in Automation of Social Promotion. 2014 AAAI Spring Symposium Series. – 2014.

33. Wille, R. Restructuring lattice theory: an approach based on hierarchies of concepts. - Ordered Sets: Dordrecht/Boston, Reidel. – 1982. - P. 445—470.

34. Medina, R., Obiedkov, S.A. (eds.). Formal Concept Analysis. 6th International Conference, ICFCA 2008, Montreal, Canada. – Springer. – 2008.

35. Newman, M.E.J., Strogatz, S., Watts, D. Random graphs with arbitrary degree distributions and their applications. Phys. Rev. E 64. - 2001.

36. Kuznetsov, S.O., Obiedkov, S., Roth, C. Reducing the representation complexity of lattice-based taxonomies. 15th Intl Conf on Conceptual Structures, ICCS 2007. - Sheffield, UK. - LNCS/LNAI. Vol. 4604. Springer. – 2007.

37. Klimushkin, M., Chetverikov, D., Novokreshchenova, A. Formal Concept Analysis of the US Blogosphere during the 2008 Presidential Campaign. 9th international session of the HSE "Baltic Practice". – Belgium. – 2009.

38. Klimushkin, M.A., Obiedkov, S.A., Roth, C. Approaches to the selection of relevant concepts in the case of noisy data. 8th International Conference, ICFCA2010, Morocco. – Springer. – 2010.

39. Ilvovsky D. A., Klimushkin M. A. FCA-based Search for Duplicate Objects in Ontologies. in: Proceedings of the Workshop Formal Concept Analysis Meets Information Retrieval / Отв. ред.: S. O. Kuznetsov, C. Carpineto, A. Napoli. Vol. 977: CEUR Workshop Proceeding, 2013.

40. Neznanov, A., Ilvovsky, D. A., Kuznetsov, S. FCART: A New FCA-based System for Data Analysis and Knowledge Discovery , in: Contributions to the 11th International Conference on Formal Concept Analysis. Dresden: Qucoza, 2013. P. 31-44.

41. Neznanov A., Ilvovsky D., Parinov A. Advancing FCA Workflow in FCART System for Knowledge Discovery in Quantitative Data. Procedia Computer Science. 2nd International Conference on Information Technology and Quantitative Management, ITQM 2014. Vol. 31. Amsterdam: ELSEVIER, 2014. P. 201-210.

42. Kuznetsov, S. O., Strok, F. V., Ilvovsky, D. A., Galitsky, B. Improving Text Retrieval Efficiency with Pattern Structures on Parse Thickets. Proceedings of the Workshop Formal Concept Analysis Meets Information Retrieval. Vol. 977. CEUR Workshop Proceeding, 2013. P. 6-21.

43. Galitsky, B., Ilvovsky, D., Kuznetsov, S. O., Strok, F. Matching sets of parse trees for answering multi-sentence questions // Proceedings of the Recent Advances in Natural Language Processing, RANLP 2013. – INCOMA Ltd., Shoumen, Bulgaria. – 2013. – P. 285–294.

44. Galitsky, B. A., Ilvovsky, D., Kuznetsov, S. O., Strok, F. Finding Maximal Common Sub-parse Thickets for Multi-sentence Search. Graph Structures for Knowledge Representation and Reasoning. Springer. – 2014. – P. 39-57.

45. Galitsky, B., Ilvovsky, D. A., Kuznetsov, S. O., Strok, F. V. Parse thicket representations of text paragraphs. Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог» В 2-х т. Т. 1: Основная программа конференции. Вып. 12 (19). М.: РГГУ, 2013. С. 134-145.

46. Ilvovsky, D. Going beyond sentences when applying tree kernels. Proceedings of the Student Research Workshop.– ACL 2014.– P. 56-63.

47. Galitsky B., Ilvovsky D., Kuznetsov S. Rhetoric map of an answer to compound queries.– ACL-IJCNLP 2015 - 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, Proceedings of the Conference. Vol. 2: Short papers. Beijing: 2015. P. 681-686.

48. Galitsky B., Ilvovsky D., Kuznetsov S. O. Text Classification into Abstract Classes Based on Discourse Structure. Proceedings of the Recent Advances in Natural Language Processing, RANLP 2015. Hissar: 2015. P. 201-207.

49. Galitsky B., Ilvovsky D., Kuznetsov S. Text integrity assessment: Sentiment profile vs rhetoric structure. Computational Linguistics and Intelligent Text Processing. 16th International Conference, CICLing 2015, Cairo, Egypt, April 14-20, 2015, Proceedings, Part II. Vol. 9042. Springer International Publishing, 2015. P. 126-139.

50. Mahalova T. N., Ilvovsky D., Galitsky B. Pattern structures for news clustering. Proceedings of the International Workshop "What can FCA do for Artificial Intelligence?" (FCA4AI at IJCAI 2015). Buenos Aires: 2015. Ch. 5. P. 35-42.

51. Strok F. V., Galitsky B., Ilvovsky D. Pattern Structure Projections for Learning Discourse Structures. Artificial Intelligence: Methodology, Systems, and Applications 16th International Conference, AIMSA 2014, Varna, Bulgaria, September 11-13, 2014. Proceedings. Vol. 8722. L., NY, Dordrecht, Heidelberg, Springer, 2014. P. 254-260.

52. Galitsky, B., Ilvovsky, D. A., Chernyak, E.L., Kuznetsov S. O. Style and Genre Classification by Means of Deep Textual Parsing. Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Международной конференции «Диалог». М.: РГГУ, 2016. С. 171-181.

53. Bhasker, B., Srikumar, K. Recommender Systems in E-Commerce. CUP. – 2010.

54. Thorsten, H., Marchand, A., Marx, P. Can Automated Group Recommender Systems Help Consumers Make Better Choices? Journal of Marketing. – 2012. – Vol. 76 (5). – P. 89–109.

55. Montaner, M., Lopez, B., de la Rosa, J. L. A Taxonomy of Recommender Agents on the Internet. Artificial Intelligence Review. – 2003. – Vol. 19 (4). – P. 285–330.

56. Taylor, A., Marcus, M., Santorini, B. The Penn treebank: an overview. Springer Netherlands. – Treebanks. – 2003. – P. 5-22.

57. Chomsky, N. Three models for the description of language. Information Theory. – IEEE Transactions. – Vol. 2 (3). – 1956 – P. 113–124.

58. Punyakanok, V., Roth, D., Yih W. The Necessity of Syntactic Parsing for Semantic Role Labeling // IJCAI-05. – 2005.

59. Domingos, P., Poon, H. Unsupervised Semantic Parsing. Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing. – 2009. – Singapore, ACL.

60. Abney, S. Parsing by Chunks. Principle-Based Parsing. Kluwer Academic Publishers. – 1991. – P. 257–278.

61. Galitsky, B., Usikov, D., Kuznetsov, S.O. Parse Thicket Representations for Answering Multi-sentence questions. 20th International Conference on Conceptual Structures, ICCS 2013. – 2013.

62. Mill, J.S. A system of logic, ratiocinative and inductive. – London, 1843.

63. Finn, V.K. On the synthesis of cognitive procedures and the problem of induction. NTI. Series 2. – 1999. – № 1–2. – P. 8–45.

64. Mitchell, T. Machine Learning. McGraw Hill. – 1997.

65. Furukawa, K. From Deduction to Induction: Logical Perspective. The Logic Programming Paradigm / eds. K. R. Apt, V. W. Marek, M. Truszczynski, D. S. Warren. – Springer, 1998.

66. Fukunaga, K. Introduction to statistical pattern recognition. Academic Press Professional Inc. – San Diego, CA, 1990.

67. Jurafsky, D., Martin, J. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. – 2008.

68. Byun, H., Lee, S. Applications of Support Vector Machines for Pattern Recognition: A Survey. Proceedings of the First International Workshop on Pattern Recognition with Support Vector Machines (SVM '02), Seong-Whan Lee and Alessandro Verri (Eds.). – 2002. – Springer-Verlag. London, UK. – P. 213–236.

69. Manning, C., Schütze, H. Foundations of Statistical Natural Language Processing. MIT Press. – 1999. – Cambridge, MA.

70. Robinson, J.A. A machine-oriented logic based on the resolution principle. Journal of the Association for Computing Machinery. – 1965. – Vol. 12. – P. 23–41.

71. Plotkin, G.D. A note on inductive generalization. B. Meltzer and D. Michie (Eds.). Machine Intelligence. – 1970. – Vol. 5. Elsevier North-Holland, New York. – P. 153–163.

72. Galitsky, B., Kuznetsov, S.O. Learning communicative actions of conflicting human agents. J. Exp. Theor. Artif. Intell. – 2008. – Vol. 20(4). – P. 277–317.

73. Galitsky B., de la Rosa, J., Dobrocsi, G. Inferring the semantic properties of sentences by mining syntactic parse trees. Data & Knowledge Engineering. – 2012. – Vol. 81–82. – P. 21–45.

74. Mann, W., Thompson, S. Rhetorical Structure Theory: Toward a Functional Theory of Text Organization. Text. 8(3). – 1988. – P. 243–281.

75. Mann, W., Matthiessen, C., Thompson, S. Rhetorical Structure Theory and Text Analysis. Discourse Description: Diverse linguistic analyses

of a fund-raising text / ed. by W. C. Mann and S. A. Thompson. – Amsterdam. – 1992. – P. 39–78.

76. Collins, M., Duffy, N. Convolution kernels for natural language. Proceedings of NIPS. – 2002. – P. 625–632.
77. Lee, H., Chang, A., Peirsman, Y., Chambers, N., Surdeanu, M., Jurafsky, D. Deterministic coreference resolution based on entity-centric, precision-ranked rules. Computational Linguistics. – 2013.
78. Zelenko, D., Aone, C., Richardella, A. Kernel methods for relation extraction. JMLR. – 2003.
79. Zhang, M., Che, W., Zhou, G., Aw, A., Tan, C., Liu, T., Li, S. Semantic role labeling using a grammar-driven convolution tree kernel. IEEE transactions on audio, speech, and language processing. – 2008. – Vol. 16 (7). – P. 1315–1329.
80. Zhang, M., Zhang, H., Li, H., Convolution Kernel over Packed Parse Forest. ACL-2010. 2010.
81. Vapnik, V. The Nature of Statistical Learning Theory. – Springer-Verlag. – 1995.
82. Searle, J. Speech acts: An essay in the philosophy of language. – Cambridge: Cambridge University. – 1969.
83. Marcu, D. From Discourse Structures to Text Summaries. Proceedings of ACL Workshop on Intelligent Scalable Text Summarization / eds. I. Mani and M. Maybury. – Madrid, 1997. – P. 82–88.
84. Hardmeier, C. Discourse in statistical machine translation. – 2014.
85. Joty, S., Nakov, P. DiscoTK: Using discourse structure for machine translation evaluation. Proceedings of the Ninth Workshop on Statistical Machine Translation. – 2014.

86. Webber B., Egg M., Kordoni V. Discourse structure and language technology //Natural Language Engineering. – 2012. – Т. 18. – №. 04. – С. 437-490.

87. Feng, V. W., Hirst, G. Patterns of local discourse coherence as a feature for authorship attribution. Literary and Linguistic Computing. – 2014. – Т. 29. – №. 2. – С. 191-198.

88. Joyce, Y., Rong, J. Discourse structure for context question answering. HLT-NAACL 2004: Workshop on Pragmatics of Question Answering, pages 23–30. – 2004.

89. Verberne, S., Boves, L., Oostdijk, N., Coppen, P. Evaluating discourse-based answer extraction for why-question answering. Proceedings of the 30th annual international ACM SIGIR conference on Research and development in information retrieval, pages 735–736. ACM. – 2007.

90. Ganter, B., Kuznetsov, S. O. Pattern Structures and Their Projections, ICCS '01. – 2001. – P. 129–142.

91. Kann, V. On the Approximability of the Maximum Common Subgraph Problem. In (STACS '92) / eds. Alain Finkel and Matthias Jantzen. – 1992. – Springer-Verlag, London, UK. – P. 377–388.

92. Sun, J., Zhang, M., Lim Tan, C. Tree Sequence Kernel for Natural Language. AAAI-25. – 2011.

93. Dean, J. Challenges in Building Large-Scale Information Retrieval Systems. URL: research.google.com/people/jeff/WSDM09-keynote.pdf.

94. Galitsky, B. Machine Learning of Syntactic Parse Trees for Search and Classification of Text. Engineering Application of AI. URL: http://dx.doi.org/10.1016/j.engappai.2012.09.017.

95. Kottmann, J., Ingersoll, G., Kosin, J., Galitsky, B. The Apache OpenNLP library. URL: http://opennlp.apache.org/documentation/1.5.3/manual/opennlp.html.

96. Haussler, D. Convolution kernels on discrete structures. 1999.

97. Mel'cuk, I. Communicative Organization in Natural Language: The Semantic-communicative Structure of Sentences. – John Benjamins Publishing, 2001.

98. Croft, B., Metzler, D., Strohman, T. Search Engines - Information Retrieval in Practice. Pearson Education. North America. 2009.

99. Salton, G., Buckley, C. Term-weighting approaches in automatic text retrieval. Information Processing & Management 24(5): 513—23. 1988.

100. John, G.H., Langley, P. Estimating Continuous Distributions in Bayesian Classifiers. In Eleventh Conference on Uncertainty in Artificial Intelligence, San Mateo, 338-45. 1995.

101. Kohavi, R. A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. International Joint Conference on Artificial Intelligence. 1137-43. 1995.

102. Moore, JS, Boyer RS. MJRTY - A Fast Majority Vote Algorithm. Automated Reasoning: Essays in Honor of Woody Bledsoe, Automated Reasoning Series, Kluwer Academic Publishers, Dordrecht, The Netherlands, pp. 105-17. 1991.

103. Yevtushenko, S.A. System of data analysis "Concept Explorer". Proceedings of the 7th national conference on Artificial Intelligence KII-2000. Russia. 2000. P. 127-134.

104. Valtchev, P., Grosser, D., Roume, C., Hacene, M.R. GALICIA: an open platform for lattices, in Using Conceptual Structures.

Contributions to the 11th Intl. Conference on Conceptual Structures (ICCS'03). – 2003. P. 241-254.

105. Becker, P., Hereth J., and Stumme G. ToscanaJ: An Open Source Tool for Qualitative Data Analysis. Workshop FCAKDD of the 15th European Conference on Artificial Intelligence (ECAI 2002). Lyon. – 2002.

106. Priss, U. FcaStone - FCA file format conversion and interoperability software. Conceptual Structures Tool Interoperability Workshop (CS-TIW). – 2008.

107. Lahcen, B., Kwuida, L. Lattice Miner: A Tool for Concept Lattice Construction and Exploration. Supplementary Proceeding of International Conference on Formal concept analysis. –2010.

108. Borza, P.V., Sabou, O., Sacarea C. OpenFCA, an open source formal concept analysis toolbox. Proc. of IEEE International Conference on Automation Quality and Testing Robotics (AQTR). – 2010. P. 1-5.

109. Lin, J. Data-Intensive Text Processing with MapReduce. intool.github.io/MapReduceAlgorithms/MapReduce-book-final.pdf. – 2013.

110. Shawe-Taylor, J., Cristianini, N. Kernel Methods for Pattern Analysis. Cambridge Univ. Press. – 2004.

111. Cumby, C., Roth, D. Kernel methods for relational learning. ICML. – 2003.

112. Moschitti, A., Pighin, D., Basili, R. Tree kernels for semantic role labeling. Comput. Linguist. 34, 2. – 2008. P. 193-224.

113. Moschitti, A. Efficient convolution kernels for dependency and constituent syntactic trees. Proceedings of ECML.– 2006.

114. Joachims, T. Making large-scale SVM learning practical. Advances in Kernel Methods. – Support Vector Learning.– 1999.

115. Severyn, A., Moschitti, A. Structural relationships for large-scale learning of answer re-ranking. SIGIR 2012.– 2012.– P.741-750.

116. Severyn, A., Moschitti, A. 2012. Fast Support Vector Machines for Convolution Tree Kernels. Data Mining Knowledge Discovery 25.– 2012.– P.325-357.

117. Aiolli, F., Da San Martino, G., Sperduti, A., Moschitti, A. Efficient Kernel-based Learning for Trees. Proceeding of the IEEE Symposium on Computational Intelligence and Data Mining, Honolulu.– 2007.

118. Zhang, D., Sun Lee., W. Question Classification using Support Vector Machines. In Proceedings of the 26th ACM SIGIR.– 2003.– P. 26-32.

119. Yang, X.F., Su, J., Chew, C.L. Kernel-based pronoun resolution with structured syntactic knowledge. COLING-ACL'2006.– 2006.

120. Zhang, H., Zhang, M., Li, H., Aw, A., Tan, C. Forest-based tree sequence to string translation model. Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL.– 2009.

121. Levy, R., Galen, A. Tregex and Tsurgeon: tools for querying and manipulating tree data structures. 5th International Conference on Language Resources and Evaluation (LREC 2006).– 2006.

122. Suzuki, J., Hirao, H., Sasaki, Y., Maeda, E. Hierarchical Directed Acyclic Graph Kernel: Methods for Structured Natural Language Data. In Proceedings of the 41th Annual Meeting of Association for Computational Linguistics (ACL).– 2003.

123. Recasens, M., de Marneffe, M., Potts, C. The Life and Death of Discourse Entities: Identifying Singleton Mentions. Proceedings of NAACL 2013. –2013.

124. Joty, S., Carenini, G., Ng, R. T. A Novel Discriminative Framework for Sentence-Level Discourse Analysis. Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, EMNLP-CoNLL'12, pages 904–915, Jeju Island, Korea. Association for Computational Linguistics. 2012.

125. Joty, S., Carenini, G., Ng, R. T., Mehdad, Y. Combining Intra- and Multi-sentential Rhetorical Parsing for Document-level Discourse Analysis. Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics, Sofia, Bulgaria. 2013.

126. Joty, S., Moschitti, A. Discriminative Reranking of Discourse Parses Using Tree Kernels. Proceedings of EMNLP 2014. 2014.

127. Hausser, R. A Computational Model of Natural Language Communication; Interpretation, Inference, and Production in Database Semantics. Springer, Berlin, Heidelberg, New York. –2006.

128. Kamp, H., Reyle, U. Introduction to Model theoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory. Kluwer Academic Publishers, Dordrecht. – 1993.

129. Carpineto, C., Romano, G. A Lattice Conceptual Clustering System and Its Application to Browsing Retrieval. Machine Learning 24(2): 95-122.– 1996.

130. Carpineto, C., Romano, G. Using concept lattices for text retrieval and mining. Formal Concept Analysis. Springer-Verlag, Berlin, Heidelberg. 161-179. 2005.

131. Priss, U. Lattice-based information retrieval. Knowl. Organ. 27, 132–142. 2000.

132. Cole II, R., Eklund, P., Stumme, G. Document Retrieval for E-Mail Search and Discovery Using Formal Concept Analysis. Applied Artificial Intelligence 17(3). –2003. P.257-280.

133. Koester, B. Conceptual Knowledge Retrieval with FooCA: Improving Web Search Engine Results with Contexts and Concept Hierarchies. Industrial Conference on Data Mining 2006. –2006. P 176-190.

134. Bron, C., Kerbosch, J. Algorithm 457: finding all cliques of an undirected graph, Commun. ACM (ACM) 16 (9). –1973. P. 575–577.

135. Vismara, P., Benoît, V. Finding Maximum Common Connected Subgraphs Using Clique Detection or Constraint Satisfaction Algorithms. Modelling, Computation and Optimization in Information Systems and Management Sciences, Springer. – 2008.

136. Messai, N., Devignes, M., Napoli, A., Smaïl-Tabbone, M. Many-Valued Concept Lattices for Conceptual Clustering and Information Retrieval. ECAI 2008. – 2008. P.127-131.

137. Zamir, O., Etzioni, O. Grouper: a dynamic clustering interface to Web search results. Computer Networks 31.11. – 1999. P.1361-1374.

138. Zeng, H., He, Q., Chen, Z., Ma, W., Ma, J. Learning to cluster web search results. Proceedings of the 27th annual international ACM SIGIR conference on Research and development in information retrieval. ACM. – 2004.

139. Cimiano, P. Ontology Learning and Population from Text: Algorithms, Evaluation and Applications. Springer-Verlag New York, Inc., Secaucus, NJ, USA. 2006.

Приложения

Приложение 1

В данном приложении приведены основные фрагменты кода (на языке Java), предназначенного для реализации работы с чащами разбора, а также с синтаксическими и расширенными группами.

Определение класса для чащ разбора. Пакет parse_thicket, файл parse_thicket.java.

```
public class ParseThicket {
	// parse trees
	private List<Tree> sentenceTrees;
	// there should be an arc for each sentence
	private List<WordWordInterSentenceRelationArc> arcs;
	// lists of nodes for each sentence
	// then list for all sentences
	private List<List<ParseTreeNode>> sentenceNodes;

	public List<Tree> getSentences() {
		return sentenceTrees;
	}

	public void setSentences(List<Tree> sentences) {
		this.sentenceTrees = sentences;
	}

	public List<WordWordInterSentenceRelationArc> getArcs() {
		return arcs;
	}

	public void setArcs(List<WordWordInterSentenceRelationArc> arcs) {
		this.arcs = arcs;
	}
```

```
	public List<List<ParseTreeNode>> getNodesThicket() {
		return sentenceNodes;
	}

	public void setNodesThicket(List<List<ParseTreeNode>> nodesThicket)
{
		this.sentenceNodes = nodesThicket;
	}

	public ParseThicket(String paragraph){
		ParseCorefsBuilder builder = ParseCorefsBuilder.getInstance();
		ParseThicket res = builder.buildParseThicket(paragraph);
		this.sentenceTrees= res.sentenceTrees;
		this.arcs = res.arcs;
	}

	public ParseThicket(List<Tree> ptTrees,
			List<WordWordInterSentenceRelationArc> barcs) {
		this.sentenceTrees= ptTrees;
		this.arcs = barcs;
	}

	public String toString(){
		return this.sentenceTrees+"\n"+this.arcs;
	}

}
```

Построение групп для чащи разбора. Пакет matching, файл PT2ThicketPhraseBuilder.

```
RhetoricStructureArcsBuilder rstBuilder = new RhetoricStructureArcsBuilder();

	/*
	 * Building phrases takes a Parse Thicket and forms phrases for each
sentence individually
	 * Then based on built phrases and obtained arcs, it builds arcs for RST
```

```
        * Finally, based on all formed arcs, it extends phrases with thicket phrases
        */

        public List<List<ParseTreeNode>> buildPT2ptPhrases(ParseThicket pt ) {
            List<List<ParseTreeNode>> phrasesAllSent = new ArrayList<List<ParseTreeNode>> ();
            Map<Integer, List<List<ParseTreeNode>>> sentNumPhrases = new HashMap<Integer, List<List<ParseTreeNode>>>();
            // build regular phrases
            for(int nSent=0; nSent<pt.getSentences().size(); nSent++){


                List<ParseTreeNode> sentence = pt.getNodesThicket().get(nSent);
                Tree ptree = pt.getSentences().get(nSent);
                //ptree.pennPrint();
                List<List<ParseTreeNode>> phrases = buildPT2ptPhrasesForASentence(ptree, sentence);
                System.out.println(phrases);
                phrasesAllSent.addAll(phrases);
                sentNumPhrases.put(nSent, phrases);


            }

            // discover and add RST arcs
            List<WordWordInterSentenceRelationArc> arcsRST =

    rstBuilder.buildRSTArcsFromMarkersAndCorefs(pt.getArcs(), sentNumPhrases, pt);

            List<WordWordInterSentenceRelationArc> arcs = pt.getArcs();
            arcs.addAll(arcsRST);
            pt.setArcs(arcs);
```

```
                List<List<ParseTreeNode>> expandedPhrases = expandTowardsThicketPhrases(phrasesAllSent, pt.getArcs(), sentNumPhrases, pt);
                return expandedPhrases;
        }

    /* Take all phrases, all arcs and merge phrases into Thicket phrases.
     * Then add the set of generalized (Thicket) phrases to the input set of phrases
     * phrasesAllSent - list of lists of phrases for each sentence
     * sentNumPhrase - map , gives for each sentence id, the above list
     * arcs - arcs formed so far
     * pt - the built Parse Thicket
     */
        private List<List<ParseTreeNode>> expandTowardsThicketPhrases(
                List<List<ParseTreeNode>> phrasesAllSent,
                List<WordWordInterSentenceRelationArc> arcs,
                Map<Integer, List<List<ParseTreeNode>>> sentNumPhrases,
                ParseThicket pt ) {
            List<List<ParseTreeNode>> thicketPhrasesAllSent = new ArrayList<List<ParseTreeNode>>();


                for(int nSent=0; nSent<pt.getSentences().size(); nSent++){
                    for(int mSent=nSent+1; mSent<pt.getSentences().size(); mSent++){
                        // for given arc, find phrases connected by this arc and add to the list of phrases
                        for(WordWordInterSentenceRelationArc arc: arcs){
                            List<List<ParseTreeNode>> phrasesFrom = sentNumPhrases.get(nSent);
                            List<List<ParseTreeNode>> phrasesTo = sentNumPhrases.get(mSent);
                            int fromIndex = arc.getCodeFrom().getFirst();
                            int toIndex = arc.getCodeTo().getFirst();
```

```
                                        if (nSent==fromIndex &&
mSent==toIndex){
                                            int sentPosFrom =
arc.getCodeFrom().getSecond();
                                            int sentPosTo =
arc.getCodeTo().getSecond();
                                            // for the given arc arc, find
phrases which are connected by it
                                            List<ParseTreeNode>
lFromFound = null, lToFound = null;
                                            for(List<ParseTreeNode>
lFrom: phrasesFrom){
                                                if (lToFound!=null)
                                                    break;
                                                for(ParseTreeNode
lFromP: lFrom){
                                                    if
(lFromP.getId()!=null &&  lFromP.getId()==sentPosFrom){
        lFromFound = lFrom;
        break;
                                                    }
                                                }
                                            }
                                            for(List<ParseTreeNode>
lTo: phrasesTo){
                                                if (lToFound!=null)
                                                    break;
                                                for(ParseTreeNode
lToP: lTo)
                                                    if
(lToP.getId()!=null && lToP.getId()==sentPosTo){
        lToFound = lTo;
        break;
                                                    }
                                            }
```

```
                                                    // obtain a thicket phrase and add it to the list
                                                    if (lFromFound!=null && lToFound!=null){

                                                        if (identicalSubPhrase(lFromFound, lToFound))
                                                            continue;

        List<ParseTreeNode> appended = append(lFromFound, lToFound);
                                                        if (thicketPhrasesAllSent.contains(appended))
                                                            continue;

        System.out.println("rel: "+arc);

        System.out.println("From "+lFromFound);

        System.out.println("TO "+lToFound);

        thicketPhrasesAllSent.add(append(lFromFound, lToFound));
                                                        //break;
                                                    }
                                                }

                                            }
                                        }
                                    }
                                    phrasesAllSent.addAll(thicketPhrasesAllSent);
                                    return phrasesAllSent;
                }

        /* check that one phrase is subphrase of another by lemma (ignoring other node properties)
         * returns true if not found different word
         */

                private boolean identicalSubPhrase(List<ParseTreeNode> lFromFound,
```

```
                        List<ParseTreeNode> lToFound) {
                for(int pos=0; pos<lFromFound.size()&& pos<lToFound.size();
pos++){
                        if
(!lFromFound.get(pos).getWord().equals(lToFound.get(pos).getWord()))
                                return false;
                }
                return true;
        }

        private List<ParseTreeNode> append(List<ParseTreeNode>
lFromFound,
                        List<ParseTreeNode> lToFound) {
                List<ParseTreeNode> appendList = new
ArrayList<ParseTreeNode>();
                appendList.addAll(lFromFound);
                appendList.addAll(lToFound);
                return appendList;
        }


        public List<List<ParseTreeNode>>
buildPT2ptPhrasesForASentence(Tree tree, List<ParseTreeNode> sentence ) {
                List<List<ParseTreeNode>> phrases;

                phrases = new ArrayList<List<ParseTreeNode>>();
                navigateR(tree, sentence, phrases);

                return phrases;
        }



    /*
     *
```

[[<1>NP'Iran':NNP], [<2>VP'refuses':VBZ, <3>VP'to':TO, <4>VP'accept':VB, <5>VP'the':DT, <6>VP'UN':NNP,

<7>VP'proposal':NN, <8>VP'to':TO, <9>VP'end':VB, <10>VP'its':PRP$, <11>VP'dispute':NN, <12>VP'over':IN, <13>VP'its':PRP$,

<14>VP'work':NN, <15>VP'on':IN, <16>VP'nuclear':JJ, <17>VP'weapons':NNS], [<3>VP'to':TO, <4>VP'accept':VB, <5>VP'the':DT,

<6>VP'UN':NNP, <7>VP'proposal':NN, <8>VP'to':TO, <9>VP'end':VB, <10>VP'its':PRP$, <11>VP'dispute':NN, <12>VP'over':IN,

<13>VP'its':PRP$, <14>VP'work':NN, <15>VP'on':IN, <16>VP'nuclear':JJ, <17>VP'weapons':NNS], [<4>VP'accept':VB,

<5>VP'the':DT, <6>VP'UN':NNP, <7>VP'proposal':NN, <8>VP'to':TO, <9>VP'end':VB, <10>VP'its':PRP$, <11>VP'dispute':NN,

<12>VP'over':IN, <13>VP'its':PRP$, <14>VP'work':NN, <15>VP'on':IN, <16>VP'nuclear':JJ, <17>VP'weapons':NNS],

[<5>NP'the':DT, <6>NP'UN':NNP, <7>NP'proposal':NN], [<8>VP'to':TO, <9>VP'end':VB, <10>VP'its':PRP$, <11>VP'dispute':NN,

<12>VP'over':IN, <13>VP'its':PRP$, <14>VP'work':NN, <15>VP'on':IN, <16>VP'nuclear':JJ, <17>VP'weapons':NNS],

[<9>VP'end':VB, <10>VP'its':PRP$, <11>VP'dispute':NN, <12>VP'over':IN, <13>VP'its':PRP$, <14>VP'work':NN, <15>VP'on':IN,

<16>VP'nuclear':JJ, <17>VP'weapons':NNS], [<10>NP'its':PRP$, <11>NP'dispute':NN], [<12>PP'over':IN, <13>PP'its':PRP$,

<14>PP'work':NN, <15>PP'on':IN, <16>PP'nuclear':JJ, <17>PP'weapons':NNS], [<13>NP'its':PRP$, <14>NP'work':NN,

<15>NP'on':IN, <16>NP'nuclear':JJ, <17>NP'weapons':NNS], [<13>NP'its':PRP$, <14>NP'work':NN],

[<15>PP'on':IN, <16>PP'nuclear':JJ, <17>PP'weapons':NNS], [<16>NP'nuclear':JJ, <17>NP'weapons':NNS]]

*

*

*/

```
private void navigateR(Tree t, List<ParseTreeNode> sentence,
        List<List<ParseTreeNode>> phrases) {
    if (!t.isPreTerminal()) {
        if (t.label() != null) {
            if (t.value() != null) {
                // if ROOT or S, returns empty
                List<ParseTreeNode> nodes = parsePhrase(t.label().value(), t.toString());
```

```
                                        nodes = assignIndexToNodes(nodes, sentence);
                                        if (!nodes.isEmpty())
                                                phrases.add(nodes);
                                        if (nodes.size()>0 && nodes.get(0).getId()==null){
                                                System.err.println("Failed alignment:"+nodes);
                                        }
                                }
                        }
                        Tree[] kids = t.children();
                        if (kids != null) {
                                for (Tree kid : kids) {
                                        navigateR(kid,sentence,  phrases);
                                }
                        }
                        return ;
                }
        }


        /* alignment of phrases extracted from tree against the sentence as a list of lemma-pos */

        private List<ParseTreeNode> assignIndexToNodes(List<ParseTreeNode> node,
                        List<ParseTreeNode> sentence) {
                if (sentence==null || sentence.size()<1)
                        return node;

                List<ParseTreeNode> results = new ArrayList<ParseTreeNode>();

                for(int i= 0; i<node.size(); i++){
                        String thisLemma = node.get(i).getWord();
```

```
                String thisPOS = node.get(i).getPos();
                String nextLemma = null, nextPOS = null;

                if (i+1<node.size()){
                        nextLemma = node.get(i+1).getWord();
                        nextPOS = node.get(i+1).getPos();
                }
                Boolean matchOccurred = false;
                int j = 0;
                for(j= 0; j<sentence.size(); j++){
                        if  (!(sentence.get(j).getWord().equals(thisLemma)
&& (sentence.get(j).getPos().equals(thisPOS))))
                                continue;
                        if (i+1<node.size() && j+1 < sentence.size() &&
nextLemma!=null
                                        &&                              !
(sentence.get(j+1).getWord().equals(nextLemma)
                                        &&
sentence.get(j+1).getPos().equals(nextPOS)))
                                continue;
                        matchOccurred = true;
                        break;
                }

                ParseTreeNode n = node.get(i);
                if (matchOccurred){
                        n.setId(sentence.get(j).getId());
                        n.setNe(sentence.get(j).getNe());
                }
                results.add(n);
        }

        try {
                if      (results!=null      &&      results.size()>1      &&
results.get(0)!=null && results.get(0).getId()!=null &&
                                results.get(1)          !=null          &&
results.get(1).getId()!=null &&  results.get(1).getId()>0){
```

```
                        ParseTreeNode p = results.get(0);
                        p.setId(results.get(1).getId()-1);
                        results.set(0, p);
                    }
                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                return results;
            }



            /*
             * [[NP":], ['(NNP':Iran)], [VP":], ['(VBZ':refuses)], [VP":], ['(TO':to)], [VP":], ['(VB':accept)], [NP":],
             * ['(DT':the)], ['(NNP':UN)], ['(NN':proposal)], [VP":], ['(TO':to)], [VP":], ['(VB':end)], [NP":],
             * ['(PRP$':its)], ['(NN':dispute)], [PP":], ['(IN':over)], [NP":], [NP":],
             * ['(PRP$':its)], ['(NN':work)], [PP":], ['(IN':on)], [NP":], ['(JJ':nuclear)], ['(NNS':weapons)], ['(.':.)]]
             *
             * [[NP":], ['(NNP':Iran)],
        [VP":], ['(VBZ':refuses)],
        [VP":], ['(TO':to)],
        [VP":], ['(VB':accept)],
          [NP":], ['(DT':the)], ['(NNP':UN)], ['(NN':proposal)],
          [VP":], ['(TO':to)], [VP":], ['(VB':end)],
          [NP":], ['(PRP$':its)], ['(NN':dispute)],
             [PP":], ['(IN':over)],
                [NP":], [NP":], ['(PRP$':its)], ['(NN':work)],
                 [PP":], ['(IN':on)],
                   [NP":], ['(JJ':nuclear)], ['(NNS':weapons)],
       ['(.':.)]]
             */
            private void navigateR1(Tree t, List<ParseTreeNode> sentence, int l,
                    List<List<ParseTreeNode>> phrases) {
```

```
                if (t.isPreTerminal()) {
                        if (t.label() != null) {
                                List<ParseTreeNode>  node  =
parsePhrase(t.toString());
                                if (!node.isEmpty())
                                        phrases.add(node);
                        }
                        return;
                } else {
                        if (t.label() != null) {
                                if (t.value() != null) {
                                        List<ParseTreeNode>  node  =
parsePhrase(t.label().value());
                                        if (!node.isEmpty())
                                                phrases.add(node);
                                }
                        }
                        Tree[] kids = t.children();
                        if (kids != null) {
                                for (Tree kid : kids) {
                                        navigateR1(kid,sentence,  l, phrases);
                                }
                        }
                        return ;
                }
        }


        protected List<ParseTreeNode> parsePhrase(String value) {
                List<ParseTreeNode> nlist = new ArrayList<ParseTreeNode>();
                if (value==null)
                        return nlist;
                if (value.equals("ROOT")|| value.equals("S"))
                        return nlist;

                String[] pos_value = value.split(" ");
```

```
            ParseTreeNode node = null;
            if (value.endsWith("P")){
                    node = new ParseTreeNode("", "");
               node.setPhraseType(value);
            } else
            if (pos_value != null && pos_value.length==2){
                    node = new ParseTreeNode(pos_value[0], pos_value[1]);
            } else {
                    node = new ParseTreeNode(value, "");
            }

            nlist.add(node);
            return nlist;
    }

    private ParseTreeNode parsePhraseNode(String value) {

            if (value.equals("ROOT")|| value.equals("S"))
                    return null;

            String[] pos_value = value.split(" ");
            ParseTreeNode node = null;
            if (value.endsWith("P")){
                    node = new ParseTreeNode("", "");
               node.setPhraseType(value);
            } else
            if (pos_value != null && pos_value.length==2){
                    node = new ParseTreeNode(pos_value[0], pos_value[1]);
            } else {
                    node = new ParseTreeNode(value, "");
            }

            return node;
    }
```

```
	public List<ParseTreeNode> parsePhrase(String value, String fullDump)
{

		List<ParseTreeNode> nlist = new ArrayList<ParseTreeNode>();
		if (value.equals("S")|| value.equals("ROOT"))
			return nlist;

		String flattened =
fullDump.replace("(ROOT","").replace("(NP","").replace("(VP","").replace("(PP","")

	.replace("(ADVP","").replace("(UCP","").replace("(ADJP","").replace("(SBAR"
,"").
			replace("(PRT", "").replace("(WHNP","").

replace("))))",")").replace(")))",")").replace("))",")")
			.replace("   ", " ").replace("  ", " ").replace("(S","")
			.replace(") (","#").replace(")  (", "#");
		String[] flattenedArr =  flattened.split("#");
		for(String term: flattenedArr){
			term = term.replace('(', ' ').replace(')',' ').trim();
			if (term!=null && term.split(" ")!=null && term.split("
").length==2){
				ParseTreeNode node = new
ParseTreeNode(term.split(" ")[1],term.split(" ")[0] );
				node.setPhraseType(value);
				nlist.add(node);
			}
		}
		return nlist;
	}

	/* recursion example */

	private StringBuilder toStringBuilder(StringBuilder sb, Tree t) {
		if (t.isLeaf()) {
			if (t.label() != null) {
				sb.append(t.label().value());
```

```
                }
                return sb;
            } else {
                sb.append('(');
                if (t.label() != null) {
                    if (t.value() != null) {
                        sb.append(t.label().value());
                    }
                }
                Tree[] kids = t.children();
                if (kids != null) {
                    for (Tree kid : kids) {
                        sb.append(' ');
                        toStringBuilder(sb, kid);
                    }
                }
                return sb.append(')');
            }
        }
```

Приложение 2

В данном приложении приведены основные фрагменты кода (на языке Java), предназначенного для нахождения и обработки кореферентных связей, риторических отношений и коммуникативных действий и построения на их основе расширенных групп.

Построение кореферентных связей. Пакет parse_thicket, файл ParseCorefsBuilder.

```
public class ParseCorefsBuilder {
    protected static ParseCorefsBuilder instance;
    private Annotation annotation;
    StanfordCoreNLP pipeline;
    CommunicativeActionsArcBuilder caFinder = new
CommunicativeActionsArcBuilder();
```

```
                /**
                 * singleton method of instantiating the processor
                 *
                 * @return the instance
                 */
                public synchronized static ParseCorefsBuilder getInstance() {
                  if (instance == null)
                    instance = new ParseCorefsBuilder();

                  return instance;
                }

              ParseCorefsBuilder(){
                    Properties props = new Properties();
                    props.put("annotators", "tokenize, ssplit, pos, lemma, ner, parse, dcoref");
                    pipeline = new StanfordCoreNLP(props);
              }

              public ParseThicket buildParseThicket(String text){
                    List<Tree> ptTrees = new ArrayList<Tree>();
                    // all numbering from 1, not 0
                    List<WordWordInterSentenceRelationArc> arcs = new ArrayList<WordWordInterSentenceRelationArc>();
                    List<List<ParseTreeNode>> nodesThicket = new ArrayList<List<ParseTreeNode>>();

                    annotation = new Annotation(text);
                    try {
                          pipeline.annotate(annotation);
                          List<CoreMap> sentences = annotation.get(CoreAnnotations.SentencesAnnotation.class);
                          if (sentences != null && sentences.size() > 0)
                          for(CoreMap sentence: sentences){
                                List<ParseTreeNode> nodes = new ArrayList<ParseTreeNode>();
```

```
                    // traversing the words in the current sentence
                    // a CoreLabel is a CoreMap with additional token-specific methods
                    Class<TokensAnnotation> tokenAnn = TokensAnnotation.class;
                    List<CoreLabel> coreLabelList = sentence.get(tokenAnn);
                    int count=1;
                    for (CoreLabel token: coreLabelList ) {
                     // this is the text of the token
                     String lemma = token.get(TextAnnotation.class);
                     // this is the POS tag of the token
                     String pos = token.get(PartOfSpeechAnnotation.class);
                     // this is the NER label of the token
                     String ne = token.get(NamedEntityTagAnnotation.class);
                     nodes.add(new ParseTreeNode(lemma, pos, ne, count));
                     count++;
                    }
                    nodesThicket.add(nodes);
                   Tree tree = sentence.get(TreeCoreAnnotations.TreeAnnotation.class);
                    ptTrees.add(tree);
                  }
              } catch (Exception e) {
                    e.printStackTrace();
              }


              // now coreferences
              Map<Integer, CorefChain> corefs = annotation.get(CorefCoreAnnotations.CorefChainAnnotation.class);
              List<CorefChain> chains = new ArrayList<CorefChain>(corefs.values());
              for(CorefChain c: chains){
               //System.out.println(c);
```

```
List<CorefMention> mentions = c.getMentionsInTextualOrder();
//System.out.println(mentions);
if (mentions.size()>1)
for(int i=0; i<mentions.size(); i++){
    for(int j=i+1; j<mentions.size(); j++){
    CorefMention mi = mentions.get(i), mj=mentions.get(j);


    int niSentence = mi.position.get(0);
    int niWord = mi.startIndex;
    int njSentence = mj.position.get(0);
    int njWord = mj.startIndex;

    ArcType arcType = new ArcType("coref-", mj.mentionType+"-
"+mj.animacy, 0, 0);

    WordWordInterSentenceRelationArc arc =
                    new WordWordInterSentenceRelationArc(new
Pair<Integer, Integer>(niSentence,niWord),
                                    new Pair<Integer,
Integer>(njSentence,njWord), mi.mentionSpan, mj.mentionSpan,
                                    arcType);
    arcs.add(arc);

    /*
    System.out.println("animacy = "+m.animacy);
    System.out.println("mention span = "+m.mentionSpan);
    System.out.println(" id = "+m.mentionID);
    System.out.println(" position = "+m.position);
    System.out.println(" start index = "+m.startIndex);
    System.out.println(" end index = "+m.endIndex);
    System.out.println(" mentionType = "+m.mentionType);
    System.out.println(" number =  = "+m.number);
    */
    }
}
```

```
                }
                List<WordWordInterSentenceRelationArc> arcsCA = buildCAarcs(nodesThicket);

                ParseThicket result = new ParseThicket(ptTrees, arcs);
                result.setNodesThicket(nodesThicket);
                return result;
            }

        private List<WordWordInterSentenceRelationArc> buildCAarcs(
                                List<List<ParseTreeNode>> nodesThicket) {
                List<WordWordInterSentenceRelationArc> arcs = new ArrayList<WordWordInterSentenceRelationArc>();

                    for(int sentI=0; sentI<nodesThicket.size(); sentI++){
                            for(int sentJ=sentI+1; sentJ<nodesThicket.size(); sentJ++){
                                List<ParseTreeNode> sentenceI = nodesThicket.get(sentI),
                                                    sentenceJ = nodesThicket.get(sentJ);
                                Pair<String, Integer[]> caI = caFinder.findCAInSentence(sentenceI);
                                Pair<String, Integer[]> caJ = caFinder.findCAInSentence(sentenceJ);
                                int indexCA1 = caFinder.findCAIndexInSentence(sentenceI);
                                int indexCA2 = caFinder.findCAIndexInSentence(sentenceJ);
                                if (caI==null || caJ==null)
                                    continue;
                                Pair<String, Integer[]> caGen = caFinder.generalize(caI, caJ).get(0);

                                ArcType arcType = new ArcType("ca",

        caGen.getFirst().toString()+printNumArray(caGen.getSecond()), 0, 0);
```

```
                                        WordWordInterSentenceRelationArc arc =
                                                new
WordWordInterSentenceRelationArc(new Pair<Integer, Integer>(sentI,indexCA1),
                                                        new Pair<Integer,
Integer>(sentJ,indexCA2), caI.getFirst(), caJ.getFirst(),
                                                        arcType);
                                arcs.add(arc);

                            }
                                        }

                return arcs;
        }

    private String printNumArray(Integer[] arr){
        StringBuffer buf = new StringBuffer();
        for(Integer i: arr){
                buf.append(Integer.toString(i)+ " ");
        }
        return buf.toString();
    }
```

Выявление риторических связей, построение на их основе и обобщение расширенных групп. Пакет rhetoric_structure, файл RhetoricStructureMarker.java.

```
    public class RhetoricStructureMarker implements IGeneralizer<Integer[]>  {
        //private static String rstRelations[] = {"antithesis", "concession",
"contrast", "elaboration"};
        List<Pair<String, ParseTreeNode[]>> rstMarkers = new
ArrayList<Pair<String, ParseTreeNode[]>>();

        public  RhetoricStructureMarker(){

                rstMarkers.add(new Pair<String, ParseTreeNode[]>("contrast",
new ParseTreeNode[]{new ParseTreeNode(",",","),  new ParseTreeNode("than",",")
}));
```

```
rstMarkers.add(new Pair<String, ParseTreeNode[]>( "antithesis", new ParseTreeNode[]{new ParseTreeNode("although",","), new ParseTreeNode("*","*") }));

rstMarkers.add(new Pair<String, ParseTreeNode[]>( "contrast", new ParseTreeNode[]{new ParseTreeNode(",",",",","), new ParseTreeNode("however","*") }));

rstMarkers.add(new Pair<String, ParseTreeNode[]>( "contrast", new ParseTreeNode[]{new ParseTreeNode("however","*"), new ParseTreeNode(",",",",","),
                                                    new ParseTreeNode("*","prp"),  }));

rstMarkers.add(new Pair<String, ParseTreeNode[]>( "elaboration", new ParseTreeNode[]{new ParseTreeNode(",",",",","), new ParseTreeNode("*","NN") }));

rstMarkers.add(new Pair<String, ParseTreeNode[]>( "elaboration", new ParseTreeNode[]{new ParseTreeNode("as","*"), new ParseTreeNode("a","*") }));


rstMarkers.add(new Pair<String, ParseTreeNode[]>("explanation", new ParseTreeNode[]{new ParseTreeNode(",",",",","), new ParseTreeNode("because",",") }));

rstMarkers.add(new Pair<String, ParseTreeNode[]>( "example", new ParseTreeNode[]{new ParseTreeNode("for","IN"), new ParseTreeNode("example","NN") }));

rstMarkers.add(new Pair<String, ParseTreeNode[]>( "contrast", new ParseTreeNode[]{new ParseTreeNode(",",",",","),  new ParseTreeNode("ye","*") }));

rstMarkers.add(new Pair<String, ParseTreeNode[]>( "contrast", new ParseTreeNode[]{new ParseTreeNode("yet","*"), new ParseTreeNode(",",",",","),
                                                    new ParseTreeNode("*","prp"),  }));


rstMarkers.add(new Pair<String, ParseTreeNode[]>( "contrast", new ParseTreeNode[]{new ParseTreeNode("yet","*"), new ParseTreeNode("i","*"),
                                                  }));


rstMarkers.add(new Pair<String, ParseTreeNode[]>( "explanation", new ParseTreeNode[]{new ParseTreeNode(",",",",","), new ParseTreeNode("where","*") }));

//as long as

rstMarkers.add(new Pair<String, ParseTreeNode[]>( "temp_sequence", new ParseTreeNode[]{/*new ParseTreeNode("as","*"),*/ new ParseTreeNode("*","RB"),
                                                  new ParseTreeNode("as","IN"),}));
```

```
rstMarkers.add(new Pair<String, ParseTreeNode[]>( "temp_sequence", new ParseTreeNode[]{/*new ParseTreeNode("as","*"),*/ new ParseTreeNode("*","VB*"),
new ParseTreeNode("until","IN"),}));

}

/* For a sentence, we obtain a list of markers with the CA word and position in the sentence
 * Output span is an integer array with start/end occurrence of an RST marker in a sentence
 * */
public List<Pair<String, Integer[]>> extractRSTrelationInSentenceGetBoundarySpan(List<ParseTreeNode> sentence){
List<Pair<String, Integer[]>> results = new ArrayList<Pair<String, Integer[]>> ();

for(Pair<String, ParseTreeNode[]> template: rstMarkers){
List<Integer[]> spanList = generalize(sentence,template.getSecond() );
if (!spanList.isEmpty())
results.add(new Pair<String, Integer[]>(template.getFirst(), spanList.get(0)));
}
return results;
}

/* Rule application in the form of generalization
 * Generalizing a sentence with a rule (a template), we obtain the occurrence of rhetoric marker
 *
 * o1 - sentence
 * o2 - rule/template, specifying lemmas and/or POS, including punctuation
 * @see opennlp.tools.parse_thicket.IGeneralizer#generalize(java.lang.Object, java.lang.Object)
 * returns the span Integer[]
 */
@Override
```

```
public List<Integer[]> generalize(Object o1, Object o2) {
        List<Integer[]> result = new ArrayList<Integer[]>();

        List<ParseTreeNode> sentence = (List<ParseTreeNode> )o1;
        ParseTreeNode[] template = (ParseTreeNode[]) o2;

        boolean bBeingMatched = false;
        for(int wordIndexInSentence=0; wordIndexInSentence<sentence.size(); wordIndexInSentence++){
            ParseTreeNode word = sentence.get(wordIndexInSentence);
            int wordIndexInSentenceEnd = wordIndexInSentence; //init iterators for internal loop
            int templateIterator=0;
            while (wordIndexInSentenceEnd<sentence.size() && templateIterator< template.length){
                ParseTreeNode tword = template[templateIterator];
                ParseTreeNode currWord=sentence.get(wordIndexInSentenceEnd);
                List<ParseTreeNode> gRes = tword.generalize(tword, currWord);
                if (gRes.isEmpty()|| gRes.get(0)==null || ( gRes.get(0).getWord().equals("*")
                        && gRes.get(0).getPos().equals("*") )){
                    bBeingMatched = false;
                    break;
                } else {
                    bBeingMatched = true;
                }
                wordIndexInSentenceEnd++;
                templateIterator++;
            }
            // template iteration is done
            // the only condition for successful match is IF we are at the end of template
            if (templateIterator == template.length){
```

```
                                result.add(new Integer[]{wordIndexInSentence, wordIndexInSentenceEnd-1});
                                return result;
                        }

                        // no match for current sentence word: proceed to the next
                }
                return result;
        }

        public String markerToString(List<Pair<String, Integer[]>> res){
                StringBuffer buf = new StringBuffer();
                buf.append("[");
                for(Pair<String, Integer[]> marker: res){
                        buf.append(marker.getFirst()+":");
                        for(int a: marker.getSecond()){
                                buf.append(a+" ");
                        }
                        buf.append (" | ");
                }
                buf.append("]");
                return buf.toString();
        }
```

Выявление, построение коммуникативных действий и обобщение расширенных групп, построенных на их основе. Пакет communicative_actions, файл CommunicativeActionsArcBuilder.

```
public class CommunicativeActionsArcBuilder implements IGeneralizer<Pair<String, Integer[]>>{

        private List<Pair<String, Integer[]>> commActionsAttr = new ArrayList<Pair<String, Integer[]>>();
        public CommunicativeActionsArcBuilder(){
```

```
commActionsAttr.add(new Pair<String, Integer[]>("agree", new Integer[]{ 1, -1, -1, 1, -1}));
commActionsAttr.add(new Pair<String, Integer[]>("accept", new Integer[]{ 1, -1, -1, 1, 1}));
commActionsAttr.add(new Pair<String, Integer[]>("explain", new Integer[]{ 0, -1, 1, 1, -1}));

commActionsAttr.add(new Pair<String, Integer[]>("suggest", new Integer[]{ 1, 0, 1, -1, -1}));
commActionsAttr.add(new Pair<String, Integer[]>("claim", new Integer[]{ 1, 0, 1, -1, -1}));

// bring-attention
commActionsAttr.add(new Pair<String, Integer[]>("bring_attention", new Integer[]{ 1, 1, 1, 1, 1}));
commActionsAttr.add(new Pair<String, Integer[]>("remind", new Integer[]{ -1, 0, 1, 1, 1}));
commActionsAttr.add(new Pair<String, Integer[]>("allow", new Integer[]{ 1, -1, -1, -1, -1}));
commActionsAttr.add(new Pair<String, Integer[]>("try", new Integer[]{ 1, 0, -1, -1, -1}));
commActionsAttr.add(new Pair<String, Integer[]>("request", new Integer[]{ 0, 1, -1, 1, 1}));
commActionsAttr.add(new Pair<String, Integer[]>("understand", new Integer[]{ 0, -1, -1, 1, -1}));

commActionsAttr.add(new Pair<String, Integer[]>("inform", new Integer[]{ 0, 0, 1, 1, -1}));
commActionsAttr.add(new Pair<String, Integer[]>("notify", new Integer[]{ 0, 0, 1, 1, -1}));
commActionsAttr.add(new Pair<String, Integer[]>("report", new Integer[]{ 0, 0, 1, 1, -1}));

commActionsAttr.add(new Pair<String, Integer[]>("confirm", new Integer[]{ 0, -1, 1, 1, 1}));
commActionsAttr.add(new Pair<String, Integer[]>("ask", new Integer[]{ 0, 1, -1, -1, -1}));
commActionsAttr.add(new Pair<String, Integer[]>("check", new Integer[]{ -1, 1, -1, -1, 1}));
```

```
                commActionsAttr.add(new Pair<String, Integer[]>("ignore", new
Integer[]{       -1,     -1,     -1,     -1,     1}));
                commActionsAttr.add(new Pair<String, Integer[]>("wait", new
Integer[]{       -1,     -1,     -1,     -1,     1}));

                commActionsAttr.add(new Pair<String, Integer[]>("convince",
new Integer[]{ 0,       1,      1,      1, -1}));
                commActionsAttr.add(new Pair<String, Integer[]>("disagree",
new Integer[]{ -1,      -1,     -1,     1,      -1}));
                commActionsAttr.add(new Pair<String, Integer[]>("appeal", new
Integer[]{       -1,     1,      1,      1,      1}));
                commActionsAttr.add(new Pair<String, Integer[]>("deny", new
Integer[]{       -1,     -1,     -1,     1,      1}));
                commActionsAttr.add(new Pair<String, Integer[]>("threaten",
new Integer[]{ -1,      1, -1,  1,      1}));

                commActionsAttr.add(new Pair<String, Integer[]>("concern",
new Integer[]{ 1,       -1, -1,  1,     1}));
                commActionsAttr.add(new Pair<String, Integer[]>("afraid", new
Integer[]{       1,      -1, -1,  1,     1}));
                commActionsAttr.add(new Pair<String, Integer[]>("worri", new
Integer[]{       1,      -1, -1,  1,     1}));
                commActionsAttr.add(new Pair<String, Integer[]>("scare", new
Integer[]{       1,      -1, -1,  1,     1}));

                commActionsAttr.add(new Pair<String, Integer[]>("want", new
Integer[]{       1,      0,      -1,     -1,     -1}));
                commActionsAttr.add(new Pair<String, Integer[]>("know", new
Integer[]{       0,      -1,     -1,     1,      -1}));
                commActionsAttr.add(new Pair<String, Integer[]>("believe",
new Integer[]{ 0,       -1,     -1,     1,      -1}));
        }

        public Pair<String, Integer[]> findCAInSentence(List<ParseTreeNode>
sentence){
                for(ParseTreeNode node: sentence){
                        for(Pair<String, Integer[]> ca: commActionsAttr){
                                String lemma = (String)ca.getFirst();
```

```
                                        // canonical form lemma is a sub-string of an actual form in parseTreeNode
                                        if (node.getWord().toLowerCase().startsWith(lemma))
                                                return ca;
                                }
                        }
                        return null;
                }

                public int findCAIndexInSentence(List<ParseTreeNode> sentence){
                        for(int index = 1; index< sentence.size(); index++){
                                ParseTreeNode node = sentence.get(index);
                                for(Pair<String, Integer[]> ca: commActionsAttr){
                                        String lemma = (String)ca.getFirst();
                                        String[] lemmas = lemma.split("_");
                                        if (lemmas==null || lemmas.length<2){
                                                if (node.getWord().toLowerCase().startsWith(lemma))
                                                        return index;
                                        } else { //multiword matching
                                                for(int indexM= index+1; indexM<sentence.size(); indexM++);//
                                        }

                                }
                        }
                        return -1;
                }


                public List<Pair<String, Integer[]>> generalize(Object o1, Object o2) {
                        List<Pair<String, Integer[]>> results = new ArrayList<Pair<String, Integer[]>>();


                        String ca1 = null, ca2=null;
```

```
if (o1 instanceof String){
	ca1 = (String)o1;
	ca2 = (String)o2;
} else {
	ca1 = ((Pair<String, Integer[]>)o1).getFirst();
	ca2 = ((Pair<String, Integer[]>)o2).getFirst();
}


// find entry for ca1
Pair<String, Integer[]> caP1=null, caP2=null;
for(Pair<String, Integer[]> ca: commActionsAttr){
	String lemma = (String)ca.getFirst();
	if (lemma.equals(ca1)){
		caP1=ca;
		break;
	}
}

// find entry for ca2
for(Pair<String, Integer[]> ca: commActionsAttr){
	String lemma = (String)ca.getFirst();
	if (lemma.equals(ca2)){
		caP2=ca;
		break;
	}
}

if (ca1.equals(ca2)){
	results.add(caP1);
} else {
	// generalization of int arrays also implements
IGeneralizer
	// we take Integer[] which is a first element of as resultant
list
```

```
                        Integer[] res = new CommunicativeActionsAttribute().
                                generalize(caP1.getSecond(),
caP2.getSecond()).get(0);
                        results.add(new Pair<String, Integer[]>("", res ));
                    }

                    return results;
                }
```

Приложение 3

В данном приложении приведены основные фрагменты кода (на языке Java), предназначенного для реализации операции сходства на графах, соответствующих чащам разбора, а также на их лингвистических проекциях.

Вычисление сходства для деревьев, входящих в чащу. Пакет matching, файл ParseTreePathMatcher.java.

```
public class ParseTreePathMatcher {

  private static final int NUMBER_OF_ITERATIONS = 2;

  private ParseTreeChunkListScorer parseTreeChunkListScorer = new
ParseTreeChunkListScorer();
  private POSManager posManager = new POSManager();
  private LemmaFormManager lemmaFormManager = new
LemmaFormManager();

  public ParseTreePathMatcher() {

  }

  public ParseTreePath generalizeTwoGroupedPhrasesOLD(ParseTreePath
chunk1,
     ParseTreePath chunk2) {
   List<String> pos1 = chunk1.getPOSs();
```

```
        List<String> pos2 = chunk1.getPOSs();

        List<String> commonPOS = new ArrayList<String>(), commonLemmas =
new ArrayList<String>();
        int k1 = 0, k2 = 0;
        Boolean incrFirst = true;
        while (k1 < pos1.size() && k2 < pos2.size()) {
          // first check if the same POS
          String sim = posManager.similarPOS(pos1.get(k1), pos2.get(k2));
          if (sim != null) {
            commonPOS.add(pos1.get(k1));
            if (chunk1.getLemmas().size() > k1 && chunk2.getLemmas().size() > k2
                &&   chunk1.getLemmas().get(k1).equals(chunk2.getLemmas().get(k2)))
{
              commonLemmas.add(chunk1.getLemmas().get(k1));
            } else {
              commonLemmas.add("*");
            }
            k1++;
            k2++;
          } else if (incrFirst) {
            k1++;
          } else {
            k2++;
          }
          incrFirst = !incrFirst;
        }

        ParseTreePath res = new ParseTreePath(commonLemmas, commonPOS, 0,
0);
        // if (parseTreeChunkListScorer.getScore(res)> 0.6)
        // System.out.println(chunk1 + "  + \n"+ chunk2 + " = \n" + res);
        return res;
      }

      // A for B => B have A
```

```
        // transforms expr { A B C prep X Y }
        // into {A B {X Y} C}
        // should only be applied to a noun phrase
        public ParseTreePath prepositionalNNSTransform(ParseTreePath ch) {
          List<String> transfPOS = new ArrayList<String>(), transfLemmas = new
ArrayList<String>();
          if (!ch.getPOSs().contains("IN"))
            return ch;
          int indexIN = ch.getPOSs().lastIndexOf("IN");

          if (indexIN < 2)// preposition is a first word - should not be in a noun
                    // phrase
            return ch;
          String Word_IN = ch.getLemmas().get(indexIN);
          if (!(Word_IN.equals("to") || Word_IN.equals("on") || Word_IN.equals("in")
             || Word_IN.equals("of") || Word_IN.equals("with")
             || Word_IN.equals("by") || Word_IN.equals("from")))
            return ch;

          List<String> toShiftAfterPartPOS = ch.getPOSs().subList(indexIN + 1,
             ch.getPOSs().size());
          List<String> toShiftAfterPartLemmas = ch.getLemmas().subList(indexIN +
1,
             ch.getLemmas().size());

          if (indexIN - 1 > 0)
            transfPOS.addAll(ch.getPOSs().subList(0, indexIN - 1));
          transfPOS.addAll(toShiftAfterPartPOS);
          transfPOS.add(ch.getPOSs().get(indexIN - 1));

          if (indexIN - 1 > 0)
            transfLemmas.addAll(ch.getLemmas().subList(0, indexIN - 1));
          transfLemmas.addAll(toShiftAfterPartLemmas);
          transfLemmas.add(ch.getLemmas().get(indexIN - 1));

          return new ParseTreePath(transfLemmas, transfPOS, 0, 0);
```

```
    }

    public ParseTreePath
generalizeTwoGroupedPhrasesRandomSelectHighestScoreWithTransforms(
      ParseTreePath chunk1, ParseTreePath chunk2) {
    ParseTreePath chRes1 =
generalizeTwoGroupedPhrasesRandomSelectHighestScore(
      chunk1, chunk2);
    ParseTreePath chRes2 =
generalizeTwoGroupedPhrasesRandomSelectHighestScore(
      prepositionalNNSTransform(chunk1), chunk2);
    ParseTreePath chRes3 =
generalizeTwoGroupedPhrasesRandomSelectHighestScore(
      prepositionalNNSTransform(chunk2), chunk1);

    ParseTreePath chRes = null;
    if (parseTreeChunkListScorer.getScore(chRes1) > parseTreeChunkListScorer
      .getScore(chRes2))
     if (parseTreeChunkListScorer.getScore(chRes1) >
parseTreeChunkListScorer
       .getScore(chRes3))
      chRes = chRes1;
     else
      chRes = chRes3;
    else if (parseTreeChunkListScorer.getScore(chRes2) >
parseTreeChunkListScorer
      .getScore(chRes3))
     chRes = chRes2;
    else
     chRes = chRes3;

    return chRes;
    }

    public ParseTreePath
generalizeTwoGroupedPhrasesRandomSelectHighestScore(
      ParseTreePath chunk1, ParseTreePath chunk2) {
```

```
List<String> pos1 = chunk1.getPOSs();
List<String> pos2 = chunk2.getPOSs();
// Map <ParseTreeChunk, Double> scoredResults = new HashMap <ParseTreeChunk,
// Double> ();
int timesRepetitiveRun = NUMBER_OF_ITERATIONS;

Double globalScore = -1.0;
ParseTreePath result = null;

for (int timesRun = 0; timesRun < timesRepetitiveRun; timesRun++) {
 List<String> commonPOS = new ArrayList<String>(), commonLemmas = new ArrayList<String>();
 int k1 = 0, k2 = 0;
 Double score = 0.0;
 while (k1 < pos1.size() && k2 < pos2.size()) {
  // first check if the same POS
  String sim = posManager.similarPOS(pos1.get(k1), pos2.get(k2));
  String lemmaMatch = lemmaFormManager.matchLemmas(null, chunk1
    .getLemmas().get(k1), chunk2.getLemmas().get(k2), sim);
  // if (LemmaFormManager.acceptableLemmaAndPOS(sim, lemmaMatch)){
  if ((sim != null)
    && (lemmaMatch == null || (lemmaMatch != null && !lemmaMatch
      .equals("fail")))) {
   // if (sim!=null){ // && (lemmaMatch!=null &&
   // !lemmaMatch.equals("fail"))){
   commonPOS.add(pos1.get(k1));
   if (chunk1.getLemmas().size() > k1 && chunk2.getLemmas().size() > k2
     && lemmaMatch != null) {
    commonLemmas.add(lemmaMatch);

   } else {
    commonLemmas.add("*");

   }
```

```
            k1++;
            k2++;
          } else if (Math.random() > 0.5) {
            k1++;
          } else {
            k2++;
          }

        }
        ParseTreePath currResult = new ParseTreePath(commonLemmas, commonPOS,
            0, 0);
        score = parseTreeChunkListScorer.getScore(currResult);
        if (score > globalScore) {
          // System.out.println(chunk1 + "  + \n"+ chunk2 + " = \n" +
          // result+" score = "+ score +"\n\n");
          result = currResult;
          globalScore = score;
        }
      }

      for (int timesRun = 0; timesRun < timesRepetitiveRun; timesRun++) {
        List<String> commonPOS = new ArrayList<String>(), commonLemmas = new ArrayList<String>();
        int k1 = pos1.size() - 1, k2 = pos2.size() - 1;
        Double score = 0.0;
        while (k1 >= 0 && k2 >= 0) {
          // first check if the same POS
          String sim = posManager.similarPOS(pos1.get(k1), pos2.get(k2));
          String lemmaMatch = lemmaFormManager.matchLemmas(null, chunk1
              .getLemmas().get(k1), chunk2.getLemmas().get(k2), sim);
          // if (acceptableLemmaAndPOS(sim, lemmaMatch)){
          if ((sim != null)
              && (lemmaMatch == null || (lemmaMatch != null && !lemmaMatch
                  .equals("fail")))) {
            commonPOS.add(pos1.get(k1));
```

```
            if (chunk1.getLemmas().size() > k1 && chunk2.getLemmas().size() > k2
                && lemmaMatch != null) {
              commonLemmas.add(lemmaMatch);
            } else {
              commonLemmas.add("*");

            }
            k1--;
            k2--;
          } else if (Math.random() > 0.5) {
            k1--;
          } else {
            k2--;
          }

        }
        Collections.reverse(commonLemmas);
        Collections.reverse(commonPOS);

        ParseTreePath currResult = new ParseTreePath(commonLemmas,
commonPOS,
            0, 0);
        score = parseTreeChunkListScorer.getScore(currResult);
        if (score > globalScore) {
          // System.out.println(chunk1 + "  + \n"+ chunk2 + " = \n" +
          // currResult+" score = "+ score +"\n\n");
          result = currResult;
          globalScore = score;
        }
      }

      // // System.out.println(chunk1 + "  + \n"+ chunk2 + " = \n" + result
      // +" score = " +
      // // parseTreeChunkListScorer.getScore(result)+"\n\n");
      return result;
```

```
  }

  public Boolean acceptableLemmaAndPOS(String sim, String lemmaMatch) {
   if (sim == null) {
    return false;
   }

   if (lemmaMatch != null && !lemmaMatch.equals("fail")) {
    return false;
   }
   // even if lemmaMatch==null
   return true;
   // if (sim!=null && (lemmaMatch!=null && !lemmaMatch.equals("fail"))){

  }
 }
```

Преобразование чащи в граф. Пакет thicket2graph, файл GraphFromPTreeBuilder.java.

```
public class GraphFromPTreeBuilder {


        public Graph<ParseGraphNode, DefaultEdge> buildGraphFromPT(ParseThicket pt){
                PrintWriter out = new PrintWriter(System.out);


                List<Tree> ts = pt.getSentences();
                ts.get(0).pennPrint(out);
                Graph<ParseGraphNode, DefaultEdge> gfragment = buildGGraphFromTree(ts.get(0));

                //ParseTreeVisualizer applet = new ParseTreeVisualizer();
                //applet.showGraph(gfragment);

                return gfragment;
```

```
	}


	private Graph<ParseGraphNode, DefaultEdge> buildGGraphFromTree(Tree tree) {
		Graph<ParseGraphNode, DefaultEdge> g =
			new SimpleGraph<ParseGraphNode, DefaultEdge>(DefaultEdge.class);
		ParseGraphNode root = new ParseGraphNode(tree,"S 0");
		g.addVertex(root);
		navigate(tree, g, 0, root);

		return g;
	}



	private void navigate(Tree tree, Graph<ParseGraphNode, DefaultEdge> g, int l, ParseGraphNode currParent) {
		//String currParent = tree.label().value()+" $"+Integer.toString(l);
		//g.addVertex(currParent);
		if (tree.getChildrenAsList().size()==1)
			navigate(tree.getChildrenAsList().get(0), g, l+1, currParent);
		else
			if (tree.getChildrenAsList().size()==0)
				return;

		for(Tree child: tree.getChildrenAsList()){
			String currChild = null;
			ParseGraphNode currChildNode = null;
			try {
				if (child.isLeaf())
					continue;
				if (child.label().value().startsWith("S"))
```

```
                                navigate(child.getChildrenAsList().get(0),
g, l+1, currParent);

                            if (!child.isPhrasal() || child.isPreTerminal())
                                currChild = child.toString()+"
#"+Integer.toString(l);
                            else
                                currChild = child.label().value()+"
#"+Integer.toString(l);
                            currChildNode = new ParseGraphNode(child,
currChild);
                            g.addVertex(currChildNode);
                            g.addEdge(currParent, currChildNode);
                        } catch (Exception e) {
                            // TODO Auto-generated catch block
                            e.printStackTrace();
                        }
                        navigate(child, g, l+1, currChildNode);
                    }
                }
```

Вычисление сходства на графах, представляющих чащи разбора. Пакет thicket2graph, файл EdgeProductBuilder.java.

```
public class EdgeProductBuilder {
    private Matcher matcher = new Matcher();
    private ParseCorefsBuilder ptBuilder =
ParseCorefsBuilder.getInstance();
    private GraphFromPTreeBuilder graphBuilder = new
GraphFromPTreeBuilder();


    public Graph<ParseGraphNode[], DefaultEdge>
        buildEdgeProduct(Graph<ParseGraphNode, DefaultEdge> g1,
Graph<ParseGraphNode, DefaultEdge> g2 ){
            Graph<ParseGraphNode[], DefaultEdge> gp =
                new SimpleGraph<ParseGraphNode[],
DefaultEdge>(DefaultEdge.class);
```

```
Set<DefaultEdge> edges1 = g1.edgeSet();
Set<DefaultEdge> edges2 = g2.edgeSet();
// build nodes of product graph
for(DefaultEdge e1:edges1){
    for(DefaultEdge e2:edges2){
        ParseGraphNode sourceE1s = g1.getEdgeSource(e1), sourceE1t = g1.getEdgeTarget(e1);
        ParseGraphNode sourceE2s = g2.getEdgeSource(e2), sourceE2t = g2.getEdgeTarget(e2);

        if (isNotEmpty(matcher.generalize(sourceE1s.getPtNodes(), sourceE2s.getPtNodes())) &&

    isNotEmpty(matcher.generalize(sourceE1t.getPtNodes(), sourceE2t.getPtNodes()))
                )
                gp.addVertex(new ParseGraphNode[] {sourceE1s, sourceE1t, sourceE2s, sourceE2t } );
    }
}

Set<ParseGraphNode[]> productVerticesSet = gp.vertexSet();
List<ParseGraphNode[]> productVerticesList = new ArrayList<ParseGraphNode[]>(productVerticesSet);
for(int i=0; i<productVerticesList.size(); i++){
    for(int j=i+1; j<productVerticesList.size(); j++){
        ParseGraphNode[] prodVertexI = productVerticesList.get(i);
        ParseGraphNode[] prodVertexJ = productVerticesList.get(j);
        if (bothAjacentOrNeitherAdjacent(prodVertexI, prodVertexJ)){
            gp.addEdge(prodVertexI, prodVertexJ);
        }
    }
}
```

```
                return gp;

        }
        /*
         * Finding the maximal clique is the slowest part
         */

        public Collection<Set<ParseGraphNode[]>> getMaximalCommonSubgraphs(Graph<ParseGraphNode[], DefaultEdge> g){
                BronKerboschCliqueFinder<ParseGraphNode[], DefaultEdge> finder =
                new BronKerboschCliqueFinder<ParseGraphNode[], DefaultEdge>(g);

            Collection<Set<ParseGraphNode[]>> cliques = finder.getBiggestMaximalCliques();
                return cliques;
        }


        private boolean bothAjacentOrNeitherAdjacent(ParseGraphNode[] prodVertexI,
                        ParseGraphNode[] prodVertexJ) {
                List<ParseGraphNode> prodVertexIlist =
                        new ArrayList<ParseGraphNode>(Arrays.asList(prodVertexI));
                List<ParseGraphNode> prodVertexJlist =
                        new ArrayList<ParseGraphNode>(Arrays.asList(prodVertexJ));
                prodVertexIlist.retainAll(prodVertexJlist);
                return (prodVertexIlist.size()==2 || prodVertexIlist.size()==4);
        }

        private boolean isNotEmpty(List<List<ParseTreeChunk>> generalize) {
                if (generalize!=null && generalize.get(0)!=null && generalize.get(0).size()>0)
                        return true;
                else
```

```
                return false;
        }
        public Collection<Set<ParseGraphNode[]>>
assessRelevanceViaMaximalCommonSubgraphs(String para1, String para2) {
                // first build PTs for each text
                ParseThicket pt1 = ptBuilder.buildParseThicket(para1);
                ParseThicket pt2 = ptBuilder.buildParseThicket(para2);
                // then build phrases and rst arcs
                Graph<ParseGraphNode, DefaultEdge> g1 =
graphBuilder.buildGraphFromPT(pt1);
                Graph<ParseGraphNode, DefaultEdge> g2 =
graphBuilder.buildGraphFromPT(pt2);

                Graph<ParseGraphNode[], DefaultEdge> gp =
buildEdgeProduct(g1, g2);
                Collection<Set<ParseGraphNode[]>> col =
getMaximalCommonSubgraphs(gp);
                return col;
                }
```

Приложение 4

В данном приложении приведены основные фрагменты кода (на языке Java), предназначенного для реализации поиска ответа на сложные вопросы с помощью вычисления сходства чащ разбора и их проекций для вопроса и потенциальных ответов.

Оценка итогового значения релевантности (score) на основе результатов операции сходства текстовых абзацев. Пакет textsimilarity, файл ParseTreeChunkListScorer.java.

```
public class ParseTreeChunkListScorer {
  // find the single expression with the highest score
  public double getParseTreeChunkListScore(
     List<List<ParseTreeChunk>> matchResult) {
    double currScore = 0.0;
    for (List<ParseTreeChunk> chunksGivenPhraseType : matchResult)
      for (ParseTreeChunk chunk : chunksGivenPhraseType) {
```

```
      Double score = getScore(chunk);
      // System.out.println(chunk+ " => score >>> "+score);
      if (score > currScore) {
        currScore = score;
      }
    }
  return currScore;
}

// get max score per phrase type and then sum up
public double getParseTreeChunkListScoreAggregPhraseType(
    List<List<ParseTreeChunk>> matchResult) {
  double currScoreTotal = 0.0;
  for (List<ParseTreeChunk> chunksGivenPhraseType : matchResult) {
    double currScorePT = 0.0;
    for (ParseTreeChunk chunk : chunksGivenPhraseType) {
      Double score = getScore(chunk);
      // System.out.println(chunk+ " => score >>> "+score);
      if (score > currScorePT) {
        currScorePT = score;
      }
    }
    // if substantial for given phrase type
    if (currScorePT > 0.5) {
      currScoreTotal += currScorePT;
    }
  }
  return currScoreTotal;
}

// score is meaningful only for chunks which are results of generalization

public double getScore(ParseTreeChunk chunk) {
  double score = 0.0;
  int i = 0;
```

```
    for (String l : chunk.getLemmas()) {
      String pos = chunk.getPOSs().get(i);
      if (l.equals("*")) {
        if (pos.startsWith("CD")) { // number vs number gives high score
                                    // although different numbers
          score += 0.7;
        } else if (pos.endsWith("_high")) { // if query modification adds 'high'
          score += 1.0;
        } else {
          score += 0.1;
        }
      } else {

        if (pos.startsWith("NN") || pos.startsWith("NP")
            || pos.startsWith("CD") || pos.startsWith("RB")) {
          score += 1.0;
        } else if (pos.startsWith("VB") || pos.startsWith("JJ")) {
          if (l.equals("get")) { // 'common' verbs are not that important
            score += 0.3;
          } else {
            score += 0.5;
          }
        } else {
          score += 0.3;
        }
      }
      i++;

    }
    return score;
  }

}
```

Переупорядочивание результатов поиска. Пакет textsimilarity, файл SearchResultsProcessor.java.

```
public class SearchResultsProcessor extends BingQueryRunner {
  private static Logger LOG = Logger
    .getLogger("opennlp.tools.similarity.apps.SearchResultsProcessor");
  private ParseTreeChunkListScorer parseTreeChunkListScorer = new ParseTreeChunkListScorer();
  ParserChunker2MatcherProcessor sm;
  WebSearchEngineResultsScraper scraper = new WebSearchEngineResultsScraper();

  /*
   * Takes a search engine API (or scraped) search results and calculates the parse tree similarity
   * between the question and each snippet. Ranks those snippets with higher
   * similarity score up
   */

  private List<HitBase> calculateMatchScoreResortHits(List<HitBase> hits,
    String searchQuery) {

   List<HitBase> newHitList = new ArrayList<HitBase>();
   sm = ParserChunker2MatcherProcessor.getInstance();

   for (HitBase hit : hits) {
    String snapshot = hit.getAbstractText().replace("<b>...</b>", ". ").replace("<span class='best-phrase'>", " ").replace("<span>", " ").replace("<span>", " ")
      .replace("<b>", "").replace("</b>", "");
    snapshot = snapshot.replace("</B>", "").replace("<B>", "")
      .replace("<br>", "").replace("</br>", "").replace("...", ". ")
      .replace("|", " ").replace(">", " ");
    snapshot += " . " + hit.getTitle();
    Double score = 0.0;
    try {
     SentencePairMatchResult matchRes = sm.assessRelevance(snapshot,
       searchQuery);
     List<List<ParseTreeChunk>> match = matchRes.getMatchResult();
```

```
      score = parseTreeChunkListScorer.getParseTreeChunkListScore(match);
      LOG.finest(score + " | " + snapshot);
    } catch (Exception e) {
      LOG.severe("Problem processing snapshot " + snapshot);
      e.printStackTrace();
    }
    hit.setGenerWithQueryScore(score);
    newHitList.add(hit);
  }
  Collections.sort(newHitList, new HitBaseComparable());

  LOG.info("\n\n ============= NEW ORDER ================= ");
  for (HitBase hit : newHitList) {
    LOG.info(hit.toString());
  }

  return newHitList;
}

public void close() {
  sm.close();
}

public List<HitBase> runSearch(String query) {


  List<HitBase> hits = scraper.runSearch(query);
  hits = calculateMatchScoreResortHits(hits, query);
  return hits;
}

public List<HitBase> runSearchViaAPI(String query) {
        List<HitBase> hits = null;
  try {
    List<HitBase> resultList = runSearch(query);
```

```
      // now we apply our own relevance filter
      hits = calculateMatchScoreResortHits(resultList, query);
    } catch (Exception e) {
      // e.printStackTrace();
      LOG.info("No search results for query '" + query);
      return null;
    }
    return hits;
  }
}
```

Приложение 5

В данном приложении приведены основные фрагменты кода (на языке Java), предназначенного для построения узорных структур и их проекций на чащах разбора и реализации алгоритма кластеризации текстов.

Построение проекции узорной структуры на чащах разбора, алгоритм AddIntent. Пакет pattern_structure, файл PhrasePatternStructure.

```
public class PhrasePatternStructure {
        int objectCount;
        int attributeCount;
        ArrayList<PhraseConcept> conceptList;
        ParseTreeMatcherDeterministic md;
        public PhrasePatternStructure(int objectCounts, int attributeCounts) {
                objectCount = objectCounts;
                attributeCount = attributeCounts;
                conceptList = new ArrayList<PhraseConcept>();
                PhraseConcept bottom = new PhraseConcept();
                md = new ParseTreeMatcherDeterministic();
                /*Set<Integer> b_intent = new HashSet<Integer>();
                for (int index = 0; index < attributeCount; ++index) {
                        b_intent.add(index);
```

```
                }
                bottom.setIntent(b_intent);*/
                bottom.setPosition(0);
                conceptList.add(bottom);
        }
        public int GetMaximalConcept(List<List<ParseTreeChunk>> intent, int
Generator) {
                boolean parentIsMaximal = true;
                while(parentIsMaximal) {
                        parentIsMaximal = false;
                        for (int parent : conceptList.get(Generator).parents) {
                                if
(conceptList.get(parent).intent.containsAll(intent)) {
                                        Generator = parent;
                                        parentIsMaximal = true;
                                        break;
                                }
                        }
                }
                return Generator;
        }
        public int AddIntent(List<List<ParseTreeChunk>> intent, int generator)
{
                System.out.println("debug");
                System.out.println("called for " + intent);
                //printLattice();
                int generator_tmp = GetMaximalConcept(intent, generator);
                generator = generator_tmp;
                if (conceptList.get(generator).intent.equals(intent)) {
                        System.out.println("at generator:" +
conceptList.get(generator).intent);
                        System.out.println("to add:" + intent);

                        System.out.println("already generated");
                        return generator;
                }
```

```
                Set<Integer> generatorParents = conceptList.get(generator).parents;
                Set<Integer> newParents = new HashSet<Integer>();
                for (int candidate : generatorParents) {
                    if (!intent.containsAll(conceptList.get(candidate).intent)) {
                    //if (!conceptList.get(candidate).intent.containsAll(intent)) {
                        //Set<Integer> intersection = new HashSet<Integer>(conceptList.get(candidate).intent);
                        //List<List<ParseTreeChunk>> intersection = new ArrayList<List<ParseTreeChunk>>(conceptList.get(candidate).intent);
                        //intersection.retainAll(intent);
                        List<List<ParseTreeChunk>> intersection = md

            .matchTwoSentencesGroupedChunksDeterministic(intent, conceptList.get(candidate).intent);
                        System.out.println("recursive call (inclusion)");
                        candidate = AddIntent(intersection, candidate);
                    }
                    boolean addParents = true;
                    System.out.println("now iterating over parents");
                    Iterator<Integer> iterator = newParents.iterator();
                    while (iterator.hasNext()) {
                        Integer parent = iterator.next();
                        if (conceptList.get(parent).intent.containsAll(conceptList.get(candidate).intent)) {
                            addParents = false;
                            break;
                        }
                        else {
                            if (conceptList.get(candidate).intent.containsAll(conceptList.get(parent).intent)) {
                                iterator.remove();
                            }
                        }
                    }
                    /*for (int parent : newParents) {
```

```
                                        System.out.println("parent = " + parent);
                                        System.out.println("candidate
intent:"+conceptList.get(candidate).intent);
                                        System.out.println("parent
intent:"+conceptList.get(parent).intent);


                                        if
(conceptList.get(parent).intent.containsAll(conceptList.get(candidate).intent)) {
                                                addParents = false;
                                                break;
                                        }
                                        else {
                                                if
(conceptList.get(candidate).intent.containsAll(conceptList.get(parent).intent)) {
                                                        newParents.remove(parent);
                                                }
                                        }
                                }*/
                                if (addParents) {
                                        newParents.add(candidate);
                                }
                        }
                        System.out.println("size of lattice: " + conceptList.size());
                        PhraseConcept newConcept = new PhraseConcept();
                        newConcept.setIntent(intent);
                        newConcept.setPosition(conceptList.size());
                        conceptList.add(newConcept);
                        conceptList.get(generator).parents.add(newConcept.position);
                        for (int newParent: newParents) {
                                if
(conceptList.get(generator).parents.contains(newParent)) {

        conceptList.get(generator).parents.remove(newParent);
                                }

        conceptList.get(newConcept.position).parents.add(newParent);
                        }
```

```
                return newConcept.position;
        }
        public void printLatticeStats() {
                System.out.println("Lattice stats");
                System.out.println("max_object_index = " + objectCount);
                System.out.println("max_attribute_index = " + attributeCount);
                System.out.println("Current concept count = " +
conceptList.size());
        }
        public void printLattice() {
                for (int i = 0; i < conceptList.size(); ++i) {
                        printConceptByPosition(i);
                }
        }
        public void printConceptByPosition(int index) {
                System.out.println("Concept at position " + index);
                conceptList.get(index).printConcept();
        }
        public                                  List<List<ParseTreeChunk>>
formGroupedPhrasesFromChunksForPara(
                        List<List<ParseTreeNode>> phrs) {
                List<List<ParseTreeChunk>>      results     =     new
ArrayList<List<ParseTreeChunk>>();
                List<ParseTreeChunk>            nps         =     new
ArrayList<ParseTreeChunk>(), vps = new ArrayList<ParseTreeChunk>(),
                                pps = new ArrayList<ParseTreeChunk>();
                for(List<ParseTreeNode> ps:phrs){
                        ParseTreeChunk ch = convertNodeListIntoChunk(ps);
                        String ptype = ps.get(0).getPhraseType();
                        if (ptype.equals("NP")){
                                nps.add(ch);
                        } else if (ptype.equals("VP")){
                                vps.add(ch);
                        } else if (ptype.equals("PP")){
                                pps.add(ch);
                        }
```

```
                }
                results.add(nps); results.add(vps); results.add(pps);
                return results;
        }
        private ParseTreeChunk convertNodeListIntoChunk(List<ParseTreeNode> ps) {
                List<String> lemmas = new ArrayList<String>(), poss = new ArrayList<String>();
                for(ParseTreeNode n: ps){
                        lemmas.add(n.getWord());
                        poss.add(n.getPos());
                }
                ParseTreeChunk ch = new ParseTreeChunk(lemmas, poss, 0, 0);
                ch.setMainPOS(ps.get(0).getPhraseType());
                return ch;
        }

}
```

Построение и фильтрация узорной структуры на чащах разбора. Пакет pattern_structure, файл LinguisticPhrasePatternStructure, класс LinguisticPhrasePatternStructure (наследует классу PhrasePatternStructure).

```
public class LinguisticPatternStructure extends PhrasePatternStructure {

        public LinguisticPatternStructure(int objectCounts, int attributeCounts) {

                super(objectCounts, attributeCounts);



                ConceptLattice cl = null;

        }
```

```
public void AddExtentToAncestors(LinkedHashSet<Integer>extent, int curNode) {
		//
		if (conceptList.get(curNode).parents.size()>0){
			for (int parent : conceptList.get(curNode).parents){
				conceptList.get(parent).addExtents(extent);
				AddExtentToAncestors(extent, parent);
			}
		}
	}

	public int AddIntent(List<List<ParseTreeChunk>> intent, LinkedHashSet<Integer>extent,int generator) {
		System.out.println("debug");
		System.out.println("called for " + intent);
		//printLattice();
		int generator_tmp = GetMaximalConcept(intent, generator);
		generator = generator_tmp;
		if (conceptList.get(generator).intent.equals(intent)) {
			System.out.println("at generator:" + conceptList.get(generator).intent);
			System.out.println("to add:" + intent);
			System.out.println("already generated");
```

```
                AddExtentToAncestors(extent, generator);
                return generator;
            }
            Set<Integer> generatorParents = conceptList.get(generator).parents;
            Set<Integer> newParents = new HashSet<Integer>();
            for (int candidate : generatorParents) {
                if (!intent.containsAll(conceptList.get(candidate).intent)) {
                    List<List<ParseTreeChunk>> intersection = md
        .matchTwoSentencesGroupedChunksDeterministic(intent, conceptList.get(candidate).intent);
                    LinkedHashSet<Integer> new_extent = new LinkedHashSet<Integer>();
        new_extent.addAll(conceptList.get(candidate).extent);
                    new_extent.addAll(extent);
                    if (intent.size()!=intersection.size()){
                        System.out.println("recursive call (inclusion)");
                        System.out.println(intent + "----" + intersection);
                        candidate = AddIntent(intersection,new_extent, candidate);
```

```
                }
            }

            boolean addParents = true;
            System.out.println("now iterating over parents");
            Iterator<Integer> iterator = newParents.iterator();
            while (iterator.hasNext()) {
                Integer parent = iterator.next();
                if
(conceptList.get(parent).intent.containsAll(conceptList.get(candidate).intent)) {
                    addParents = false;
                    break;
                }
                else {
                    if
(conceptList.get(candidate).intent.containsAll(conceptList.get(parent).intent)) {
                        iterator.remove();
                    }
                }
            }
            if (addParents) {
                newParents.add(candidate);
```

```
                    }
                }
                System.out.println("size of lattice: " + conceptList.size());
                PhraseConcept newConcept = new PhraseConcept();
                newConcept.setIntent(intent);

                LinkedHashSet<Integer> new_extent = new
LinkedHashSet<Integer>();
                new_extent.addAll(conceptList.get(generator).extent);
                new_extent.addAll(extent);
                newConcept.addExtents(new_extent);

                newConcept.setPosition(conceptList.size());
                conceptList.add(newConcept);
                conceptList.get(generator).parents.add(newConcept.position);
                conceptList.get(newConcept.position).childs.add(generator);
                for (int newParent: newParents) {
                    if
(conceptList.get(generator).parents.contains(newParent)) {

        conceptList.get(generator).parents.remove(newParent);

        conceptList.get(newParent).childs.remove(generator);
```

```
                }

conceptList.get(newConcept.position).parents.add(newParent);
                conceptList.get(newParent).addExtents(new_extent);
                AddExtentToAncestors(new_extent, newParent);

conceptList.get(newParent).childs.add(newConcept.position);
            }
            return newConcept.position;
        }

        public void printLatticeExtended() {
            for (int i = 0; i < conceptList.size(); ++i) {
                printConceptByPositionExtended(i);
            }
        }

        public void printConceptByPositionExtended(int index) {
            System.out.println("Concept at position " + index);
            conceptList.get(index).printConceptExtended();
        }
```

```
public int [][] toContext(int extentCardinality){

        int newAttrCount = conceptList.size();
        ArrayList<PhraseConcept> cList = new ArrayList<PhraseConcept>();
        cList.addAll(conceptList);
        boolean run = true;
        int k=0;
        while (run && k<conceptList.size()){
            if (conceptList.get(k).intent.size() == attributeCount){
                if (conceptList.get(k).extent.size() == 0)
                    for (Integer i:conceptList.get(k).parents)
                        cList.remove(i);
                cList.remove(k);
                run=false;
            }
            else
                k+=1;
        }

        run = true;
        k=0;
```

```
                while (run && k<=newAttrCount){
                        if (cList.get(k).extent.size()==0)
                                k++;
                        run = false;
                }
                newAttrCount = cList.size();
                Set<Integer> nodeExtend;
                int [][] binaryContext = new int[extentCardinality][newAttrCount];
                for (int j = 0; j<newAttrCount; j++){
                        nodeExtend = cList.get(j).extent;
                        for (Integer i: nodeExtend){
                                binaryContext[i][j]=1;
                        }
                }
                return binaryContext;
        }

        public void logStability(){
                int min_delta = -1, delta = -1;
                float sum = 0;
                for (int i = 0; i < conceptList.size(); ++i) {
```

```
                    min_delta = Integer.MAX_VALUE;
                    sum = 0;
                    PhraseConcept pc = conceptList.get(i);
                    Set<Integer> childs = pc.childs;
                    for (Integer j: childs) {
                        delta = pc.extent.size() -
conceptList.get(j).extent.size();
                        if (delta<min_delta)
                            min_delta = delta;
                        sum += Math.pow(2, -delta);
                    }
                    pc.intLogStabilityBottom=-
(Math.log(sum)/Math.log(2.0));
                    pc.intLogStabilityUp = min_delta;
                }
            }
        }
```

Приложение 6

В данном приложении приведены основные фрагменты кода (на языке Java), применявшегося для обучения на текстовых абзацах.

Обучение и классификация на лесе регулярных деревьев разбора. Пакет kernel_interface, файл MultiSentenceKernelBasedSearchResultsProcessor.

```
public class MultiSentenceKernelBasedSearchResultsProcessor extends
MultiSentenceSearchResultsProcessor{
```

```
		private static Logger LOG = Logger

	.getLogger("opennlp.tools.similarity.apps.MultiSentenceKernelBasedSearchResultsProcessor");


		private WebSearchEngineResultsScraper scraper = new WebSearchEngineResultsScraper();
		protected Matcher matcher = new Matcher();
		private ParseTreeChunkListScorer parseTreeChunkListScorer = new ParseTreeChunkListScorer();
		protected BingQueryRunnerMultipageSearchResults bingSearcher = new BingQueryRunnerMultipageSearchResults();
		private SnippetToParagraph snp = new SnippetToParagraph();
		private TreeKernelRunner tkRunner = new TreeKernelRunner();


		protected final float lower_threshold = (float) 0.2;
		protected final float upper_threshold = (float) 0.8;
		protected final float ratio = (float) 0.4; //соотношение обучающей и тестовой выборки


		private String path;
		public void setKernelPath (String path){
			this.path=path;
		}
		protected static final String modelFileName = "model.txt";


		protected static final String trainingFileName = "training.txt";


		protected static final String unknownToBeClassified = "unknown.txt";


		protected static final String classifierOutput = "classifier_output.txt";


		protected static final String detailedOutput = "\\Answers\\answers_test.csv";


		protected static final String detailedLearningOutput = "\\Answers\\answers_learn.csv";
```

```
public List<HitBase> runSearchViaAPI(String query) {
    List<HitBase> hits = null;
    List<String[]> output = new ArrayList<String[]>();
    String[] sent;
    String[] fullQuery = query.split("!!!");
    try {
        List<HitBase> resultList = bingSearcher.runSearch(fullQuery[2], 100);//100
        // now we apply our own relevance filter
        //hits = calculateMatchScoreResortHits(resultList, query);

        hits = resultList;
        //once we applied our re-ranking, we set highly ranked as positive set, low-rated as negative set
        //and classify search results from the middle
        //training set is formed from original documents for the search results,
        // and 10 of these search results from the middle are classified
        //true for snippets
        hits = filterOutIrrelevantHitsByTreeKernelLearning(hits, fullQuery[2], false);//true for snippets

        //copying results to the List<String[]>
        sent = new String[2];
        sent[1] = fullQuery[0] + " " + fullQuery[1];

        output.add(new String[] {""});
        output.add(new String[] {""});
        output.add(sent);
        sent = new String[2];
        sent[1] = fullQuery[2];
        output.add(sent);
        output.add(new String[] {""});

        for(HitBase h : hits){
```

```
                                        sent = new String[2];
                                        sent[0] = "  " + h.getGenerWithQueryScore().toString() + " ";

                                        output.add(sent);

                                        sent = new String[1];
                                        sent[0] = "page content = " + h.getPageContent();
                                        output.add(sent);
                                        sent = new String[1];
                                        sent[0] = "orig sent = " + h.getOriginalSentences().toString();
                                        output.add(sent);
                                        sent = new String[1];
                                        sent[0] = "title = " + h.getTitle();
                                        output.add(sent);
                                        sent = new String[1];
                                        sent[0] = "abstract = " + h.getAbstractText();
                                        output.add(sent);
                                        output.add(new String[] {""});
                                }
                                //appending results to the report
                                ProfileReaderWriter.appendReport(output, path+detailedOutput);

                        } catch (Exception e) {
                                e.printStackTrace();
                                LOG.info("No search results for query '" + fullQuery[2]);
                                return null;
                        }


                        return hits;
                }

                private List<HitBase> filterOutIrrelevantHitsByTreeKernelLearning(
```

```
                    List<HitBase> hits, String query, Boolean onlySnippets)
{
                    List<HitBase> newHitList = new ArrayList<HitBase>();
                    List<HitBase> newHitListTraining = new ArrayList<HitBase>();
                    // form the training set from original documents. Since search
results are ranked, we set the top-20 as positive set,
                    //and the bottom-20 as negative set.
                    // after re-classification, being re-ranked, the search results might
end up in a different set
                    List<String[]> treeBankBuffer = new ArrayList<String[]>();
                    List<String[]>      treeBankClassifyBuffer      =      new
ArrayList<String[]>();
                    String snippet;
                    Random rnd = new Random();


                    int count = 0;
                    int flag = 0;
                    for (HitBase hit : hits) {
                        count++;
                        flag = 0;
                        // if orig content has been already set in HIT object, ok;
otherwise set it
                        if
(!(hit.getUrl().matches(".*\\.(doc[xm]|doc|DOC[XM]|DOC|jar|JAR|XLS|xls)$"))){
                        //if (hit.getUrl().contains("ATTACHMENT01")){
                            String searchResultText;
                            if (!(onlySnippets)) {
                                searchResultText = hit.getPageContent();
                                if (searchResultText == null) {
                                    try {
                                        String[]
pageSentsAndSnippet = formTextForReRankingFromHit(hit);
                                        searchResultText      =
pageSentsAndSnippet[0];

    hit.setPageContent(searchResultText);
                                    } catch (Exception e) {
```

```
                                        // skip if we are not able to
fetch page content
                                        e.printStackTrace();
                                        flag = -1;
                                    } catch (NoClassDefFoundError
e1) {
                                        e1.printStackTrace();
                                        flag = -1;
                                    } catch (NoSuchMethodError e1) {
                                        // skip if we are not able to
fetch page content
                                        e1.printStackTrace();
                                        flag = -1;
                                    }
                                }
                            } else {//getting snippets
                                searchResultText = hit.getAbstractText();
                                snippet                                 =
searchResultText.replace("<b>...</b>",  ".  ").replace("<span  class='best-phrase'>",  "
").replace("<span>", " ").replace("<span>", " ")
                                        .replace("<b>",
"").replace("</b>", "");
                                snippet      =      snippet.replace("</B>",
"").replace("<B>", "")
                                        .replace("<br>",
"").replace("</br>", "").replace("...", ". ")
                                        .replace("|",                "
").replace(">", " ").replace(". .", ". ");
                                searchResultText =  hit.getTitle() + " " +
snippet;
                            }
                        if (flag != -1) {
                            //newHitList.add(hit);
                            if (count <= (int) (hits.size() * lower_threshold)
                                    ||  count  >=  (int)  (hits.size()  *
upper_threshold)) {

                treeBankBuffer.addAll(formTreeKernelStructure(
```

```
                                                searchResultText, count,
hits));
                                        }
                                        //middle 10 are used for classification
                                        if (count > (int) (hits.size() * lower_threshold)
                                                && count < (int) (hits.size() *
upper_threshold)
                                                && rnd.nextFloat() <= ((float) (1) -
upper_threshold + lower_threshold)
                                                        * ratio /
(upper_threshold - lower_threshold)) {
                                                treeBankClassifyBuffer

        .addAll(formTreeKernelClassifyStructure(searchResultText));
                                                newHitList.add(hit);
                                        }
                                }
                         }
                        }
                        // write the lits of samples to a file
                        ProfileReaderWriter.writeReport(treeBankBuffer,
path+trainingFileName, ' ');

                        //add examples to log
                        /*String[] arrQuery = new String[1];
                        arrQuery[0] = "Query = " + query;
                        treeBankBuffer.add(0, arrQuery);
                        treeBankBuffer.add(new String[] {" "});
                        ProfileReaderWriter.appendReport(treeBankBuffer,
detailedLearningOutput);
                        // build the model
                        tkRunner.runLearner(path, trainingFileName, modelFileName);*/

                        // now we preparing the same answers to be classifies in/out
                        /*treeBankBuffer = new ArrayList<String[]>();
                        for (HitBase hit : newHitList) {
                                // not original docs now but instead a snippet
```

```
                String searchResultTextAbstr = hit.getAbstractText();
                String snippet = searchResultTextAbstr.replace("<b>...</b>", ". ").replace("<span class='best-phrase'>", " ").replace("<span>", " ").replace("<span>", " ")
                        .replace("<b>", "").replace("</b>", "");
                snippet = snippet.replace("</B>", "").replace("<B>", "")
                        .replace("<br>", "").replace("</br>", "").replace("...", ". ")
                        .replace("|", " ").replace(">", " ").replace("..", ". ");
                snippet =  hit.getTitle() + " " + snippet;

                ParseThicket pt = matcher.buildParseThicketFromTextWithRST(snippet);
                        //hit.getPageContent());
                List<Tree> forest = pt.getSentences();
                // we consider the snippet as a single sentence to be classified
                if (forest.size()>0){
                    treeBankBuffer.add(new String[] {"0 |BT| "+forest.get(0).toString()+ " |ET|"});
                    newHitListReRanked .add(hit);
                }

            }*/

            ProfileReaderWriter.writeReport(treeBankClassifyBuffer, path+unknownToBeClassified, ' ');
            tkRunner.runClassifier(path, unknownToBeClassified, modelFileName, classifierOutput);

            // read classification results
            List<String[]> classifResults = ProfileReaderWriter.readProfiles(path+classifierOutput, ' ');
            HitBase h;
            // iterate through classification results and set them as scores for hits
            //newHitList = new ArrayList<HitBase>();
```

```
                for(int i=0; i < newHitList.size() && i < classifResults.size() ; i++){
                        String scoreClassif = classifResults.get(i)[0];
                        float val = Float.parseFloat(scoreClassif);

                        h = newHitList.get(i);
                        h.setGenerWithQueryScore(((double) val));

                        newHitList.set(i, h);
                }

                // sort by SVM classification results
                Collections.sort(newHitList, new HitBaseComparable());
                System.out.println("\n\n ============= NEW ORDER ================= ");
                for (HitBase hit : newHitList) {
                        if (!(onlySnippets)){
                         System.out.println(hit.getOriginalSentences().toString() + " => "+hit.getGenerWithQueryScore());
                         System.out.println("page content = "+hit.getPageContent());
                        }
                        System.out.println("title = "+hit.getAbstractText());
                        System.out.println("snippet = "+hit.getAbstractText());
                        System.out.println("match = "+hit.getSource());
                }

                return newHitList;

        }

        protected List<String[]> formTreeKernelStructure(String searchResultText, int count, List<HitBase> hits) {
                List<String[]> treeBankBuffer = new ArrayList<String[]> ();
                try {
                        // if from the top of ranked docs, then positive, if from the bottom - negative
                        String posOrNeg = null;
```

```
                        if (count<=(int) (hits.size() * lower_threshold))
                                posOrNeg=" 1 ";
                        else if (count >= (int) (hits.size() * upper_threshold))
                                posOrNeg=" -1 ";
                        else
                                posOrNeg=" 0 ";//middle for classification
                        // form the list of training samples
                        if (posOrNeg != " 0 "){
                                // get the parses from original documents, and form the training dataset
                                ParseThicket pt = matcher.buildParseThicketFromTextWithRST(searchResultText);
                                List<Tree> forest = pt.getSentences();

                                String[] sent = new String[1];
                                sent[0] = posOrNeg;
                                for(Tree t: forest){
                                        //treeBankBuffer.add(new String[] {posOrNeg+" |BT| "+t.toString()+ " |ET|"});
                                        sent[0] = sent[0] + " |BT| " + t.toString();
                                }
                                if (sent[0] == posOrNeg){
                                        sent[0] += "|BT| |ET|";
                                }
                                else {
                                        sent[0] += " |ET|";
                                }
                                treeBankBuffer.add(sent);
                        }
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                return treeBankBuffer;
        }
```

```
protected List<String[]> formTreeKernelClassifyStructure(String searchResultText) {
        List<String[]> treeBankBuffer = new ArrayList<String[]> ();
        try {
            ParseThicket pt = matcher.buildParseThicketFromTextWithRST(searchResultText);
            List<Tree> forest = pt.getSentences();
            String[] sent = new String[1];
            sent[0] = " 0 ";
            for(Tree t: forest){
                //treeBankBuffer.add(new String[] {" 0 " + |BT| "+t.toString()+ " |ET|"});
                sent[0] = sent[0] + " |BT| " + t.toString();
            }
            if (sent[0] == " 0 "){
                sent[0] += "|BT| |ET|";
            }
            else {
                sent[0] += " |ET|";
            }
            treeBankBuffer.add(sent);

        } catch (Exception e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        return treeBankBuffer;
    }
```

Обучение и классификация на лесе расширенных деревьев. Пакет kernel_interface, файл MultiSentenceKernelBasedExtendedForestSearchResultsProcessor (данный класс наследует класс, описанный в предыдущем файле).

```
public class MultiSentenceKernelBasedExtendedForestSearchResultsProcessor extends MultiSentenceKernelBasedSearchResultsProcessor{
    private static Logger LOG = Logger
```

```
.getLogger("opennlp.tools.similarity.apps.MultiSentenceKernelBasedExtendedForestSearchResultsProcessor");

protected TreeExtenderByAnotherLinkedTree treeExtender = new TreeExtenderByAnotherLinkedTree();


protected List<String[]> formTreeKernelStructure(String searchResultText, int count, List<HitBase> hits) {
    List<String[]> treeBankBuffer = new ArrayList<String[]> ();
    try {
        // if from the top of ranked docs then positive, if from the bottom - negative
        String posOrNeg = null;
        if (count<=(int) (hits.size() * lower_threshold))
            posOrNeg=" 1 ";
        else if (count >= (int) (hits.size() * upper_threshold))
            posOrNeg=" -1 ";
        else
            posOrNeg=" 0 ";//middle for classification
        // form the list of training samples
        if (posOrNeg != " 0 "){
            // get the parses from original documents, and form the training dataset
            ParseThicket pt = matcher.buildParseThicketFromTextWithRST(searchResultText);
            List<String> extendedTreesDump = treeExtender.buildForestForCorefArcs(pt);

            String[] sent = new String[1];
            sent[0] = posOrNeg;
            List<Tree> forest = pt.getSentences();
            for(Tree t: forest){
                //treeBankBuffer.add(new String[] {" 0 " + |BT| "+t.toString()+ " |ET|"});
                sent[0] = sent[0] + " |BT| " + t.toString();
            }
            //adding trees with semantic arcs
            for(String t: extendedTreesDump){
```

```
                                        //treeBankBuffer.add(new String[]
{posOrNeg+" |BT| "+t.toString()+ " |ET|"});
                                        sent[0] = sent[0] + " |BT| " + t;
                                }
                                if (sent[0] == posOrNeg){
                                        sent[0] += "|BT| |ET|";
                                }
                                else {
                                        sent[0] += " |ET|";
                                }
                                treeBankBuffer.add(sent);
                        }
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                return treeBankBuffer;
        }

        protected List<String[]> formTreeKernelClassifyStructure(String
searchResultText) {
                List<String[]> treeBankBuffer = new ArrayList<String[]> ();
                try {
                        ParseThicket pt =
matcher.buildParseThicketFromTextWithRST(searchResultText);
                        List<String> extendedTreesDump =
treeExtender.buildForestForCorefArcs(pt);
                        String[] sent = new String[1];
                        sent[0] = " 0 ";
                        List<Tree> forest = pt.getSentences();
                        for(Tree t: forest){
                                //treeBankBuffer.add(new String[] {" 0 " + |BT|
"+t.toString()+ " |ET|"});
                                sent[0] = sent[0] + " |BT| " + t.toString();
                        }
                        //adding trees with semantic arcs
```

```
                    for(String t: extendedTreesDump){

                        sent[0] = sent[0] + " |BT| " + t;
                    }
                    if (sent[0] == " 0 "){
                        sent[0] += "|BT| |ET|";
                    }
                    else {
                        sent[0] += " |ET|";
                    }
                    treeBankBuffer.add(sent);

                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
                return treeBankBuffer;
            }
```

Подготовка запросов на основе YahooAnswers. Пакет apps, файл YahooAnswersMiner.

```
public class YahooAnswersMiner extends BingQueryRunner{

    private static final Logger LOG = Logger

.getLogger("opennlp.tools.similarity.apps.YahooAnswersMiner");
    private int page = 0;
    private static final int hitsPerPage = 50;

    public List<HitBase> runSearch(String query) {
        aq.setAppid(BING_KEY);
        aq.setQuery("site:answers.yahoo.com "+
```

```
                    query);
                aq.setPerPage(hitsPerPage);
                aq.setPage(page);

                aq.doQuery();
                List<HitBase> results = new ArrayList<HitBase> ();
                AzureSearchResultSet<AzureSearchWebResult> ars = aq.getQueryResult();

                for (AzureSearchWebResult anr : ars){
                    HitBase h = new HitBase();
                    h.setAbstractText(anr.getDescription());
                    h.setTitle(anr.getTitle());
                    h.setUrl(anr.getUrl());
                    results.add(h);
                }
                page++;

                return results;
            }

            public List<HitBase> runSearch(String query, int totalPages) {
                int count=0;
                List<HitBase> results = new ArrayList<HitBase>();
                while(totalPages>page*hitsPerPage){
```

```
                List<HitBase> res = runSearch(query);
                results.addAll(res);
                if (count>10)
                        break;
                count++;
        }
        return results;
}
```

Приложение 7

В данном приложении приведены ключевые фрагменты кода (на языке Python), использовавшегося для реализации метода выявления тождественных денотатов и проведения экспериментов на прикладной онтологии и сгенерированных формальных контекстах.

Генерация данных. Файл create_context.py.

```
import random as r
import math
def create_context(path, n = 0, m = 0, dubls = 0):
  if n == 0:
    n = r.randint(1000, 10000)
  if m == 0:
    m = r.randint(1000, 20000)
  if dubls == 0:
    dubls = r.randint(1, 1000)

  print('|G| - |D| = ' + str(n))
  print('|M| = ' + str(m))
  print('|D| = ' + str(dubls))
  f = open(path, 'w')
  st = ''
  table = []
  for i in range(n):
```

```
    attrs_num = min([int(-1 * math.log(0.1 * r.random()) + int(r.random())), m])
    attrs = []
    while len(attrs) < attrs_num:
      attr = round(r.random()**3 * m)
      if attr not in attrs:
        attrs.append(attr)
    table.append(attrs)
    for attr in attrs:
      st += 'X' + str(i) + ';M' + str(attr) + '\n'
  f.write(st)
  st = ''
  for i in range(dubls):
    dub = r.randint(0, n)
    attrs = table[dub]
    nattrs = []
    for attr in attrs:
      if r.random() < 0.8:
        nattrs.append(attr)
    while r.random() < 0.4:
      attr = r.randint(0, m)
      if attr not in nattrs:
        nattrs.append(attr)
    table.append(nattrs)
    for attr in nattrs:
      st += 'Y' + str(i) + 'X' + str(dub) + ';M' + str(attr) + '\n'
  f.write(st)
  f.close()
```

Интерфейс для работы с АФП. Файл fca_lib.py.

```
class Concept:
    def __init__(self, obj, attr, main_index, aux_index1 = None, aux_index2 =
None):
        self.objects = obj
        self.attributes = attr
        self.main_index = main_index
```

```
            self.aux_index1 = aux_index1
            self.aux_index2 = aux_index2
            self.stability = None

        def get_stability(self):
            return self.stability

    class Concepts:
        def __init__(self, concepts=[]):
            self.concepts = sorted(concepts[:], key = lambda x: -x.main_index)

        def __init__(self, path, p):
            concepts = []
            concepts_file = open(path, encoding = 'utf-8')
            blocks = concepts_file.read().split('\n\n')[1:-1]

            for block in blocks:
              subblocks = block.split('\n')
              attrs = subblocks[0].split(': ')[1].split(', ')
              objs = subblocks[1].split(': ')[1].split(', ')
              objs = [obj for obj in objs if len(obj) > 0]
              main_index = float(subblocks[3].split(':')[1]) + p *
float(subblocks[4].split(':')[1])
              concepts.append(Concept(objs, attrs, main_index))
            self.list = sorted(concepts, key = lambda x: -x.main_index)
            print('Concepts are loaded: ' + str(len(self.list)))
            concepts_file.close()

    class Context():
        def __init__(self, path):
            context_file = open(path, encoding = 'utf-8')
            self.obj_attr = {}
            self.attr_obj = {}
            for line in context_file:
              words = line.strip().split(';')
```

```
            obj = words[0]
            attr = words[1]
            self.obj_attr.setdefault(obj,[])
            self.obj_attr[obj].append(attr)
            self.attr_obj.setdefault(attr,[])
            self.attr_obj[attr].append(obj)
        self.objects_num = len(self.obj_attr)
        self.attributes_num = len(self.attr_obj)
        print('Context is loaded:\n\tObjects: ' + str(self.objects_num) +
              '\n\tAttributes: ' + str(self.attributes_num))
        context_file.close()
```

Подборка оптимального коэффициента. Файл find_best_koef.py.

```
import fca_lib as fl

def calc_map(rang):
  rights = []
  k = 1
  for c in rang.list:
    if len(set([int(obj.split('X')[1]) for obj in c.objects])) == 1:
      rights.append(k)
      k += 1
    else:
      rights.append(0)
  return sum([rights[i] / (i + 1) for i in range(len(rights))]
             ) / len([1 for r in rights if r > 0])


context_file_name = input('context:')
#context_file_name = 'context10.txt'
```

```
context = fl.Context(context_file_name)

points = [t * 0.05 for t in range(20)] + [t**2 for t in range(1, 10)]

res = []

for p in points:

  concepts = fl.Concepts('AggregatedIndex2.txt', p)

  res.append([-1 * calc_map(concepts), p])

for x in sorted(res):

  print(round(x[1],2), -1 * round(x[0], 6))
```

Выявление понятий, образующих тождественные денотаты. Файл select_dublicates.py.

```
import fca_lib as fl


def solve(concept):
  return concept.main_index > 2


def control(dubls):
  f = 0
  for i in range(len(dubls) - 1):
    for j in range(i + 1, len(dubls)):
      if len(dubls[i] & dubls[j]) > 0:
        if f == 0:
          print('Error! Groups have common objects:')
          f = 1
        print('\t groups ' + str(i + 1) + ' and ' + str(j + 1))
  if f == 0:
    print('Errors are not found')


def all_subsets(objs):
  for i in range(2 ** len(objs)):
    bins = bin(i)[2:]
    bins = '0' * (len(objs) - len(bins)) + bins
```

```
    sub = [objs[i] for i in range(len(objs)) if bins[i] == '1']
    yield sub

def galua(attrs, context):
  objs = set(context.obj_attr.keys())
  for attr in attrs:
    objs &= set(context.attr_obj[attr])
  return list(objs)

def stability(concepts, context):
  for c in concepts.list:
    if len(c.attributes) > 32:
      print('I can\'t do it!')
      break
    ch = 0
    zn = 2 ** len(c.attributes)
    for s in all_subsets(c.attributes):
      if len(galua(s, context)) == len(c.objects):
        ch += 1
    c.stability = ch * 1.0 / zn



context_file_name = input('context:')
#context_file_name = 'context.txt'
context = fl.Context(context_file_name)
concepts = fl.Concepts('AggregatedIndex2.txt', 4)

stab_concepts = fl.Concepts('AggregatedIndex2.txt', 4)
stability(stab_concepts, context)
stab_concepts.list.sort(key = fl.Concept.get_stability)



## Analyze ----------------------------------------------------------------
```

```
        def classification(concepts, s = solve):
          dubls = []
          for c in concepts.list:
            c.aux_index1 = 1
          for i in range(len(concepts.list)):
            c = concepts.list[i]

            auto_solve = len([d for d in dubls if len(d & set(c.objects)) == len(c.objects)])
> 0
            auto_solve = auto_solve or c.aux_index1 == 0
            if auto_solve:
              continue

            if s(c):
              relevant_dubls = [k for k in range(len(dubls)) if len(dubls[k] &
set(c.objects)) > 0]
              if len(relevant_dubls) == 0:
                dubls.append(set(c.objects))
              elif len(relevant_dubls) == 1:
                dubls[relevant_dubls[0]] |= set(c.objects)
              else:
                new_dubls = set(c.objects)
                for k in relevant_dubls:
                  new_dubls |= dubls[k]
                dubls = [dubls[r] for r in range(len(dubls)) if r not in relevant_dubls]
                dubls.append(new_dubls)
            else:
              for q in range(i + 1, len(concepts.list)):
                if (set(concepts.list[q].objects) & set(c.objects)) == set(c.objects):
                  concepts.list[q].aux_index1 = 0
          return dubls

        def ham_classification(p):
          pairs = []
          objs = [obj for obj in context.obj_attr.keys()]
          for i in range(len(objs) - 1):
```

```
    for j in range(i + 1, len(objs)):
      if len(set(context.obj_attr[objs[i]]) & set(context.obj_attr[objs[j]])) > p:
        pairs.append([objs[i], objs[j]])
  dubls = []
  for pair in pairs:
    relevant_dubls = [k for k in range(len(dubls)) if len(dubls[k] & set(pair)) > 0]
    if len(relevant_dubls) == 0:
      dubls.append(set(pair))
    elif len(relevant_dubls) == 1:
      dubls[relevant_dubls[0]] |= set(pair)
    else:
      new_dubls = set(pair)
      for k in relevant_dubls:
        new_dubls |= dubls[k]
      dubls = [dubls[r] for r in range(len(dubls)) if r not in relevant_dubls]
      dubls.append(new_dubls)
  return dubls

def ham_classification2(p):
  pairs = []
  objs = [obj for obj in context.obj_attr.keys()]
  for i in range(len(objs) - 1):
    for j in range(i + 1, len(objs)):
      if (len(set(context.obj_attr[objs[i]]) | set(context.obj_attr[objs[j]])) -
          len(set(context.obj_attr[objs[i]]) & set(context.obj_attr[objs[j]]))) < p:
        pairs.append([objs[i], objs[j]])
  dubls = []
  for pair in pairs:
    relevant_dubls = [k for k in range(len(dubls)) if len(dubls[k] & set(pair)) > 0]
    if len(relevant_dubls) == 0:
      dubls.append(set(pair))
    elif len(relevant_dubls) == 1:
      dubls[relevant_dubls[0]] |= set(pair)
    else:
      new_dubls = set(pair)
```

```
      for k in relevant_dubls:
        new_dubls |= dubls[k]
      dubls = [dubls[r] for r in range(len(dubls)) if r not in relevant_dubls]
      dubls.append(new_dubls)

  return dubls


## 1. Range test:  our index, stability
print('Test 1...')
def calc_map(rang):
  rights = []
  k = 1
  for c in rang.list:
    if len(set([int(obj.split('X')[1]) for obj in c.objects])) == 1:
      rights.append(k)
      k += 1
    else:
      rights.append(0)

  return sum([rights[i] / (i + 1) for i in range(len(rights))]
             ) / len([1 for r in rights if r > 0])


all_concepts_cnt = len(concepts.list)
right_concepts = [c for c in concepts.list
                  if len(set([int(obj.split('X')[1]) for obj in c.objects])) == 1]
right_concepts_ctn = len(right_concepts)
dubls_cnt = 0
for obj in context.obj_attr.keys():
  if 'Y' in obj:
    dubls_cnt += 1

covered_objects = set()
for c in right_concepts:
```

```
  covered_objects |= set([obj for obj in c.objects if 'Y' in obj])
covered_objects_cnt = len(covered_objects)

fstat_range = 'stat_range.txt'
fstat = open(fstat_range, 'a')
fstat.write(str(all_concepts_cnt) + '\t' +
            str(right_concepts_ctn) + '\t' +
            str(round(calc_map(concepts), 6)) + '\t' +
            str(round(calc_map(stab_concepts), 6)) + '\t' +
            str(dubls_cnt) + '\t' +
            str(covered_objects_cnt) + '\t' +
            str(round(covered_objects_cnt / dubls_cnt, 6)) + '\n')
fstat.close()


## 2. Classification test: our index, stability, Hamming
print('Test 2...')
dubls_num = 0
objs_num = 0
for obj in context.obj_attr.keys():
  if 'Y' in obj:
    dubls_num += 1
  else:
    objs_num += 1


def classif_quality(dubls):
  dubls_found = 0
  error_dubls = 0
  links_num = 0
  if len(dubls) == 0:
    return 0, 1
  for d in dubls:
    originals = [int(obj.split('X')[1]) for obj in d]
    dubls_found += len(originals) - len(set(originals))
```

```
    error_dubls += len(set(originals)) - 1
    links_num += len(originals) - 1
  return dubls_found / dubls_num, dubls_found / links_num


stab_points = [0.999999 - (t/20) for t in range(20)]
our_points = [6 - (t/5) for t in range(20)]
ham2_points = [t + 0.5 for t in range(11)]
ham_points = [11.5 - t for t in range(11)]
stab_cl_quality = []
our_cl_quality =[]
ham_cl_quality = []
ham2_cl_quality = []
print('  Stability...')
for p in stab_points:
  dubls = classification(stab_concepts, lambda x: x.stability > p)
  stab_cl_quality.append(classif_quality(dubls))
print('  Our index...')
for p in our_points:
  dubls = classification(concepts, lambda x: x.main_index > p)
  our_cl_quality.append(classif_quality(dubls))
print('  Hamming...')
for p in ham_points:
  dubls = ham_classification(p)
  ham_cl_quality.append(classif_quality(dubls))
print('  Hamming2...')
for p in ham2_points:
  dubls = ham_classification2(p)
  ham2_cl_quality.append(classif_quality(dubls))

##fres = open('bublicates-' + context_file_name, 'w')
##print('finish!!!\n\nDublicates (' + str(len(dubls)) + '):')
##control(dubls)
##k = 0
##for d in dubls:
```

```
##  k += 1
##  #print(d)
##  st = 'Dublicates group ' + str(k) + ':'
##  for obj in d:
##    st += ' ' + str(obj)
##  fres.write(st + '\n')
##


fstat = open('quality_stats.txt', 'a')
dubls_num = 0
objs_num = 0
for obj in context.obj_attr.keys():
  if 'Y' in obj:
    dubls_num += 1
  else:
    objs_num += 1
st = str(objs_num) + '\t' + str(dubls_num) + '\n'
st1, st2 = '', ''
for i in range(len(our_cl_quality)):
  st1 += str(round(our_cl_quality[i][0], 6)) + '\t'
  st2 += str(round(our_cl_quality[i][1], 6)) + '\t'
st += st1[:-1] + '\n' + st2[:-1] + '\n'
st1, st2 = '', ''
for i in range(len(stab_cl_quality)):
  st1 += str(round(stab_cl_quality[i][0], 6)) + '\t'
  st2 += str(round(stab_cl_quality[i][1], 6)) + '\t'
st += st1[:-1] + '\n' + st2[:-1] + '\n'
st1 = ''
st2 = ''
for i in range(len(ham_cl_quality)):
  st1 += str(round(ham_cl_quality[i][0], 6)) + '\t'
  st2 += str(round(ham_cl_quality[i][1], 6)) + '\t'
st += st1[:-1] + '\n' + st2[:-1] + '\n'
st1 = ''
```

```
st2 = ''
for i in range(len(ham2_cl_quality)):
  st1 += str(round(ham2_cl_quality[i][0], 6)) + '\t'
  st2 += str(round(ham2_cl_quality[i][1], 6)) + '\t'
st += st1[:-1] + '\n' + st2[:-1] + '\n'

fstat.write(st)
fstat.close()



##fstat.write(str(context.objects_num - dubls_num) + '\t' +
##            str(dubls_num) + '\t' +
##            str(dubls_found) + '\t' +
##            str(round(dubls_found / dubls_num, 6)) + '\t' +
##            str(round(dubls_found / links_num, 6)) + '\t' +
##            str(error_dubls) + '\t' +
##            str(error_dubls / (context.objects_num - dubls_num)) + '\n')
##fstat.close()
```