ИСП Российская Академия наук
Институт Системного Программирования

ISSN 2079-8156 (Print)
ISSN 2220-6426 (Online)

Труды Института Системного Программирования

Том 25

Москва 2013

ИСП **Российская Академия наук**
Институт Системного Программирования

Труды Института Системного Программирования

Том 25

Под редакцией
академика РАН В.П. Иванникова

Москва 2013

УДК004.45

Труды Института системного программирования: Том 25.
/Под ред. Академика РАН В.П. Иванникова/ – М.: ИСП РАН, 2013.

В двадцать пятом томе Трудов Института системного программирования РАН публикуются статьи, написанные сотрудниками ИСП РАН и других организаций. В этих статьях описываются результаты исследований, выполненных во второй половине 2013 г.

ISSN 2079-8156 (Print)

Содержание

П р е д и с л о в и е

В 25-м томе Трудов Института системного программирования РАН публикуются статьи, написанные сотрудниками ИСП РАН и других организаций. В этих статьях описываются результаты исследований, выполненных во второй половине 2013 г.

Пять из двенадцати статей 25-го тома посвящены разным аспектам анализа исходного кода программ. В статье С.П. Вартанова и А.Ю. Герасимова «Применение динамического анализа для поиска дефектов в программах на языке Java» рассматриваются методы анализа программ и описывается применение этих методов для автоматизации задачи поиска ошибок в программном обеспечении. Подробно обсуждается метод динамического анализа программ на основе инструментации, отслеживания потока помеченных данных и генерации наборов условий для автоматического построения входных данных. Описывается прототип инструмента, реализующего такой подход, представлены результаты его применения для анализа набора программ на языке Java.

Статья М.К. Ермакова и А.Ю. Герасимова «Avalanche: применение параллельного и распределенного динамического анализа программ для ускорения поиска дефектов и уязвимостей» посвящена подходу к уменьшению времени динамического анализа программ за счет применения параллельных и распределенных вычислений при проверке выполнимости ограничений, а также в процессе динамического анализа программ. Приводятся результаты практического применения данного подхода, реализованного в рамках инструмента динамического анализа Avalanche.

Н.Г. Зельцер в статье «Поиск повторяющихся фрагментов исходного кода при автоматическом рефакторинге» обсуждает возможность совмещения автоматического рефакторинга с обнаружением повторяющихся фрагментов исходного кода для программ на языках C/C++. Предлагается классификация программных клонов с точки зрения дальнейшего применения к ним автоматического рефакторинга. Для каждого выделенного типа клонов описывается способ их поиска. Анализируются недостатки существующих инструментов и демонстрируется, что предложенные методы в соответствующих случаях работают корректно.

В статье Н.Л. Луговского и С.В. Сыромятникова «Применение языка KAST для преобразования исходного кода и автоматического исправления дефектов» описывается расширение языка KAST для решения задачи

трансформации исходного кода. Обсуждаются некоторые существующие подходы к трансформации исходного кода и показываются преимущества использования языка KAST.

Статья «Подход к обнаружению ошибок несоответствия типов в коде на динамических языках программирования» И.Е. Бронштейна посвящена обнаружению дефектов в программах, написанных на динамических языках программирования. Демонстрируется, что большинство существующих средств не в состоянии обнаруживать целый класс дефектов: ошибки несоответствия типов. Предлагается подход к выводу типов для динамических языков, а также к реализации обнаружителей дефектов на его основе.

К тематике тестирования программных средств на основе использования формальных моделей относится статья И.С. Захарова и др. «Моделирование окружения драйверов устройств операционной системы Linux». В статье отмечается, что верификация драйвера в комбинации с исходным кодом сердцевины ядра операционной системы не представляется возможной из-за сложности и объема получаемого кода. В качестве решения этой проблемы предлагаются методы моделирования окружения драйверов на основе π исчисления Р.Милнера и трансляции π модели окружения в программу на языке Си, которая при связывании с исходным кодом драйвера описывает с точки зрения инструментов статической верификации те же сценарии работы драйвера, что и реальное окружение драйвера в операционной системе.

Следующие четыре статьи затрагивают тематику управления данными и их анализа. В статье С.Д. Кузнецова и Н.А. Мендковича «Оценка эффективности минимизации ограничений запросов к СУБД» описываются усовершенствованные алгоритмы лексической оптимизации запросов. Эти алгоритмы обнаруживают избыточные элементы в условии выборки запроса и удаляют их. Представлены результаты применения этих оптимизационных методов и их влияние на скорость обработки запросов.

Статья В.А. Золотова и В.А. Семенова «Исследование и развитие метода декомпозиции для анализа больших пространственных данных» посвящена развитию метода декомпозиции для индексации, поиска и анализа больших пространственных данных. Главное внимание уделяется алгоритмам, основанным на регулярных октальных деревьях и обеспечивающим эффективное решение ряда вычислительных задач. Алгоритмы, в частности, применимы для моделирования сложных динамических пространственно-трехмерных сцен с объектами, имеющими протяженные границы. На основе вероятностного анализа выводятся оценки сложности, которые обобщают и

улучшают известные результаты и служат теоретическим обоснованием для применения алгоритмов к более широкому классу приложений.

В статье Д.Г. Федоренко и Н.А. Астраханцева «Автоматическое извлечение новых концептов предметно-специфичных терминов» описывается способ распознавания предметно-специфичных терминов, которые присутствуют в текущей базе знаний, но выражают отсутствующие в ней концепты. Разработанный метод может быть применен к неформальным базам знаний, поскольку требует только вычисления семантической близости между концептами и статистики встречаемости терминов в корпусе документов. Экспериментальная проверка показывает, что разработанный алгоритм превосходит существующие подходы, а также позволяет повысить точность разрешения лексической многозначности.

Антон Коршунов и др. в статье «Определение демографических атрибутов пользователей микроблогов» предлагают метод автоматического определения демографических атрибутов пользователей социальных сетей по текстам их сообщений и информации из профилей. Метод основан на алгоритме машинного обучения, его отличительными особенностями являются полностью автоматическое построение исходного набора данных для обучения и тестирования, а также поддержка широкого набора языков и демографических атрибутов. Экспериментальные исследования показали высокое качество результатов определения пола, возраста и семейного положения пользователя для наиболее популярных языков: английского, русского, немецкого, французского, итальянского и испанского.

Последние две статьи посвящены теоретическим основам компьютерной науки. В статье А.В. Шокурова «Нахождение корней систем алгебраических уравнений с помощью базиса Гребнера» описывается и обосновывается алгоритм нахождения решения системы алгебраических уравнений над полем k для идеалов нулевой размерности, если задан базис Гребнера идеала этой системы для определения лексикографического порядка на термах от ее переменных. Полученное решение лежит в алгебраическом замыкании основного поля. Приведен пример системы алгебраических уравнений, имеющей единственное решение в основном поле, а ее общее число решений экспоненциально относительно описания этой системы.

Наконец, в статье А.В. Воеводина и С.А. Косяченко «Оптимальное упорядочение конфликтующих объектов и задача коммивояжера» представлена постановка задачи оптимального упорядочения конфликтующих объектов и ее связь с задачей коммивояжера (Travelling Salesman Problem, TSP). Задача оптимального упорядочения конфликтующих объектов

возникает в социологии, при анализе графов в социальных сетях, при размещении рекламных заказов в сетях СМИ. Описаны используемые авторами на практике быстрые алгоритмы решения этой и связанных с ней задач. Также рассмотрена задача TSP с разреженной матрицей штрафов. Для задач TSP с ленточной и блочно-диагональной матрицами найдены необходимые и достаточные условия и построены точные алгоритмы, при которых достигается нулевое минимальное значение целевой функции задачи. Предложены эффективные алгоритмы для произвольных разреженных матриц. Приведены результаты аналитических и численных исследований сложности разработанных алгоритмов и точности решения, а также рекомендации по применению алгоритмов для решения задач подобного типа.

Академик РАН В.П. Иванников

Применение динамического анализа для поиска дефектов в программах на языке Java

Вартанов С. П., Герасимов А. Ю.[1]
svartanov@ispras.ru, agerasimov@ispras.ru

Аннотация. В статье рассматриваются методы анализа программ и описывается практика применения данных методов для автоматизации задачи поиска ошибок в программном обеспечении. Подробно рассмотрен метод динамического анализа программ на основе инструментации, отслеживания потока помеченных данных и генерации наборов условий для автоматического построения входных данных. Рассмотрены особенности проведения подобного анализа для приложений, написанных на языке Java. Приведено описание прототипа инструмента, реализующего описанные подходы, представлены результаты его применения для анализа набора программ на языке Java и оценка полученных результатов.

Ключевые слова: итеративный динамический анализ программ; автоматический поиск ошибок; анализ Java программ.

1. Введение

На сегодняшний день в области разработки программного обеспечения одной из важных задач является гарантия качества конечных продуктов. Под качеством понимается совокупность характеристик программного обеспечения, а именно: надёжность, сопровождаемость, практичность, эффективность, мобильность и функциональность. В любой достаточно сложной программной системе присутствуют разного рода ошибки. Поскольку программное обеспечение используется и в критических сферах человеческой деятельности, наличие блокирующих или критических ошибок в программах может приводить к катастрофическим последствиям.

1 Работа проводится при финансовой поддержке Российского фонда фундаментальных исследований, номер проекта 11-07-00466-а

Поэтому неотъемлемой частью процесса разработки программ становится этап обнаружения ошибок. Среди различных методов поиска в программах дефектов можно выделить ручное и автоматическое обнаружение ошибок.

Общая структура статьи имеет следующий вид. Во введении даются общие сведения о предпосылках задачи проведения динамического анализа, даётся краткая аргументация выбранных подходов. Во второй части более подробно рассматриваются выбранные методы, исходя из особенностей языка Java. В третьей части описывается созданный прототип и использованные при его создании решения. Четвёртая, и заключительная, часть подводит итог исследования и демонстрирует основные результаты, полученные при помощи прототипа.

1.1 Автоматический поиск дефектов

Остановимся на рассмотрении случая, при котором использование понимания логики работы приложения невозможно или нежелательно из-за значительных временных затрат. При этом также будем считать, что известен исходный, исполняемый или интерпретируемый код приложения.

Подходы к решению задачи автоматического поиска дефектов традиционно разделяют на две группы — методы статического и методы динамического анализа. В случае статического анализа поиск возможных ошибок осуществляется без запуска исследуемого приложения, например, по исходному коду.

При этом следует подчеркнуть следующие характерные особенности статического анализа:

- Потенциально возможен полный анализ всего приложения. При этом возможно, что на любых входных данных реально выполняется лишь небольшая часть кода приложения.
- Возможны ложные срабатывания.

При обнаружении дефекта возникает, во-первых, проблема проверки истинности обнаруженного дефекта, и, во-вторых, проблема воспроизведения найденного дефекта при запуске программы на определённых входных данных.

1.2 Динамический анализ

Динамический анализ, подразумевающий запуск приложения на исполнение, в свою очередь, характеризуется следующими принципами:

- Для запуска программы требуются некоторые входные данные.
- Ложные срабатывания практически отсутствуют (исключением в данном случае может быть, например, утверждение о зацикливании

программы после истечения определённого временного интервала, в течение которого программа не завершается).

Основным достоинством динамического анализа можно считать отсутствие проблемы воспроизведения дефектов. Обширный опыт использования методов обеих групп показывает, что проведение динамического анализа, обеспечивающего как можно более полное покрытие исследуемой программы, требует значительных затрат времени.

Для методов статического анализа эта проблема выражается в меньшей степени, что во многом способствует более активному развитию новых подходов в этой области. Тем не менее, особенности динамического анализа, в частности, отсутствие проблемы воспроизведения дефектов и отсутствие ложных срабатываний, являются достаточными основаниями для предоставления времени и ресурсов для использования этого способа выявления ошибок.

1.3 Инструментация кода

Одним из возможных методов проведения динамического анализа является инструментация анализируемой программы. Под инструментацией программы понимается частичное изменение, при котором она (или её часть, которую необходимо исследовать) сохраняет свою функциональность, но также производит дополнительные действия, целью которых является извлечение информации о состоянии программы в ходе её выполнения, проверка отдельных значений и т. д.

Например, в инструментированную программу могут быть добавлены утверждения, которые должны быть верны всегда, когда исследуемая программа находится в заданной точке выполнения. Однако такой подход требует знания логики программы и понимания того, какие состояния программы являются допустимыми. Это означает, что его нельзя использовать для автоматического анализа.

Основным недостатком применения инструментации в процессе анализа программы являются временные потери при работе программы, возникающие из-за внедренных дополнительных инструментирующих инструкций.

1.4 Входные данные

Поскольку при динамическом анализе программы определяющим являются результаты её запуска, важно понимать, от чего зависят эти результаты. В самом простом случае поведение программы зависит от заранее фиксированных входных данных, например, файлов или аргументов командной строки.

Сложнее дело обстоит с интерактивными приложениями, поскольку в таком случае задача описания последовательности событий, которая бы полностью определяла поведение системы, становится нетривиальной. Это верно, например, для приложений с графическим интерфейсом, веб-приложений и т. д.

Для простоты изложения далее будем исходить из того, что анализируемая программа не является интерактивной и её выполнение полностью определяется конечным набором входных данных. В этом случае возникает возможность влиять на выполнение программы путём изменения входной информации.

В свете описанных ограничений рассмотрим промежуточную задачу: определить набор входных данных, на которых выполнение программы пойдёт по заранее определённому пути. Поскольку входная информация является дискретной, она может быть представлена в виде некоторой булевой формулы. В описанных условиях, любая используемая в ходе выполнения программы переменная либо имеет константное значение, либо значение, целиком зависящее от входных данных и пути выполнения программы, предшествующего использованию этой переменной.

Можно показать, что любой путь выполнения программы определяется набором условных переходов и выбором веток, следующих за ними. Таким образом, путь зависит от некоторого набора логических ограничений на значения используемых в этих условных переходах переменных, а значит полностью определяется некоторым булевым ограничением на входные данные.

Задача поиска входных данных, необходимых для того, чтобы программа прошла по заданному пути, может быть сведена к задаче проверки выполнимости булевых формул [1] с помощью решателей (solver).

1.5 Инвертирование условий и обнаружение дефектов

Рассмотрим более конкретно, как методы решения булевых формул вместе с вычислением входных данных для инвертирования условий можно применять для обнаружения ошибок в программах (удовлетворяющих указанным ограничениям).

Предполагается, что существует средство, способное после запуска программы на конкретных входных данных определить, произошла ли ошибка, обнаружена ли уязвимость или зафиксирована утечка памяти. Указанным средством может быть, например, инструмент Valgrind [5] для программ, представленных в виде набора процессорных инструкций, либо Java-машина для интерпретируемого ею байт-кода, которая в случае обнаружения ошибки выбрасывает исключение в поток ошибок.

Тогда сперва программа запускается на начальных входных данных, которые могут быть корректными (то есть соответствовать ожидаемому формату входного файла программы или протоколу) так и не корректными. Затем происходит анализ пути, по которому прошло вычисление. Для каждой вершины пути, соответствующей условному переходу, производится попытка изменить входные данные таким образом, чтобы при запуске на них вычисление программы в данной вершине пошло по альтернативному пути (имеется в виду, что в новом пути вычисления, исходящее из данной вершины ребро будет отлично от того, что было в предыдущем пути) [4].

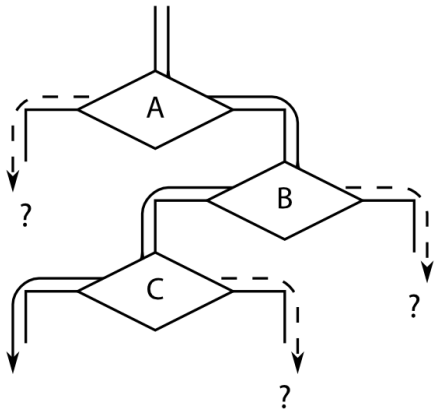


Рис. 1: Инвертирование условий

Очевидно, что такие входные данные удастся подобрать не всегда. Если, входные данные удалось обнаружить для нескольких условных переходов, на следующем шаге происходит выбор такого набора входных данных, который позволит наиболее эффективно продолжить анализ программы. В качестве примера может быть использована метрика, которая позволит проанализировать как можно большее количество ещё не проанализированных рёбер графа потока управления или тех рёбер, на которых вероятность возникновения ошибки наиболее высока.

С помощью описанного метода можно

- производить целенаправленный обход дерева путей выполнения,
- проверять наличие ошибок на заранее определённом пути выполнения.

2. Анализ программ на языке Java

Описанный выше метод проведения динамического анализа может быть применён для анализа широкого класса императивных языков: от анализируемого кода требуется строгая детерминированность выполнения последовательности действий, включая условные переходы.

В этой статье мы рассмотрим применение описанного подхода к программам на языке Java. Java — императивный объектно-ориентированный язык высокого уровня. Поскольку Java обладает средствами работы с многопоточностью, средствами генерации псевдослучайных последовательностей и прочими элементами, вносящими индетерминизм в работу программы, остановимся на подмножестве программ, не использующих данные средства. Таким образом, будем считать, что анализируемая программа имеет в качестве входных данных ограниченный набор информации и результат работы программы зависит исключительно от входного набора. Стоит отметить, что в силу широкой распространённости многопоточных приложений, таким ограничениям удовлетворяет лишь небольшой класс программ, однако зачастую описанным свойством обладают значительные части программ, требующих анализа, либо существуют простые преобразования, не нарушающие функциональности, которые приводят алгоритм в соответствие с требованиями детерминизма.

2.1 Инструментация

Для выполнения на виртуальной машине программа на языке Java транслируется в байт-код — промежуточное интерпретируемое представление программы в виде набора инструкций. Как видно из названия, любой код операции обязан умещаться в один байт, и, следовательно, число возможных операций не превышает 256.

В соответствии с описанным выше подходом, для извлечения информации о ходе выполнения программы будет применяться инструментация, т. е. изменение анализируемой программы, с целью сбора информации о работе программы.

Существует два основных способа проведения инструментации: статический и динамический. Динамическая инструментация заключается в проведении действий по изменению программы в ходе её выполнения, непосредственно перед интерпретацией. Основным достоинством динамической инструментации является то, что изменяются только действительно выполняемые части программы, т. е. не происходит избыточного изменения кода. Однако, в свете поставленной задачи, запуск программы должен происходить неоднократно, что должно приводить к повторной инструментации частей программы каждый раз, когда они будут выполняться.

Применительно к программам на языке Java это означает внесение изменений в класс-файлы (файлы, содержащие байт-код) анализируемой программы, включая класс-файлы используемых библиотек. В большинстве случаев в программе используется только некоторое подмножество классов и методов для каждого класса из подключаемых библиотек. Поэтому, прежде чем приступить к инструментации, необходимо определить это подмножество.

2.1.1 Определение списка методов для инструментации

Существуют два способа определения списка методов классов, для которых необходимо проводить инструментацию:

- Определять статически по исходному байт-коду и до запуска основного цикла программы.
- Динамически инструментировать методы, которые встретились в трассе, на каждой итерации основного цикла программы.

Какой из двух методов работает быстрее, зависит от особенностей анализируемого приложения. Влияние динамического способа на производительность программы и анализа будет выше в том случае, если в приложении в большинстве случаев используется только небольшая часть определённых в нём методов. Например, если в приложении реализованы несколько не связанных функциональностей, выбор одной из которых происходит в начале выполнения.

Однако этот метод не будет достаточно эффективным для использования в инструменте, поскольку после изменения входных данных методы, которые не вызывались при начальных входных данных, могут попасть в трассу. В этом случае придётся строить список классов и методов заново на каждой новой итерации проведения анализа и, в случае обнаружения новых используемых методов, проводить их инструментацию.

Для реализации в инструменте был выбран второй способ — статическое определение списка методов по коду программы. Этот метод имеет меньшее влияние на производительность программы и анализа в случае, если достаточно большая часть методов приложения выполняется при любых входных данных. Кроме того, уменьшается влияние на производительность самого процесса инструментации кода программы.

Построение списка методов, которые могут быть использованы, происходит итеративно. Изначально в список добавляются все методы из основного класса и считаются непомеченными. На каждом шаге итеративного анализа просматривается байт-код непомеченных методов из списка и в список непомеченных добавляются все методы, вызовы которых встретились. Просмотренные методы помечаются как инструментированные. Анализ завершается либо когда в списке не останется непомеченных методов, либо когда будет достигнуто ограничение числа шагов итеративного анализа.

Отдельно требуется добавить в список все методы инициализации классов, потому что их вызовы могут отсутствовать в байт-коде других классов, но они могут вызываться не явно самой средой выполнения программы.

2.2 Генерация условий на входные данные

Способ инструментации байт-кода программы полностью зависит от того набора информации, которую необходимо получить о ходе выполнения программы. С точки зрения обработки информации, можно выделить два основных способа инструментации: обработку по ходу выполнения программы и обработку после выполнения программы. Первый подход означает обработку информации в момент её получения, не дожидаясь окончания работы анализируемого приложения. Во втором случае в процессе выполнения программы происходит лишь сбор информации в некоторую трассу, которая будет проанализирована после завершения выполнения. Первый подход является более гибким, поскольку в ходе сбора информации можно оперировать уже извлечёнными данными и производить на их основе оптимизацию сбора оставшихся данных, однако может дополнительно увеличивать время выполнения программы, что может быть нежелательно.

Для применения описанного подхода будем использовать первый способ — обработку данных в процессе выполнения программы. Необходимо произвести инструментацию программы таким образом, чтобы в ходе выполнения программы производилось отслеживание помеченных данных (то есть таких данных, значение которых зависит от входных данных программы), определялись условные переходы, зависящие от помеченных данных, и производилось построение наборов условий для их инвертирования.

Условия, добавляемые в текущий набор, генерируются для всех операций с памятью (перемещение, копирование, арифметические операции и т. д.), если в них участвуют помеченные данные.

2.2.1 Условные переходы

Особые действия выполняются, если в трассе встретилась операция условного перехода. В терминах байт-кода это операции, в которых переход происходит в зависимости от значений операндов — это могут быть значения регистров, вершины стека, или передаваемых аргументов, в зависимости от виртуальной машины, на которой байт-код будет исполняться.

Если эта операция зависит от помеченных данных, сгенерированный на данный момент набор условий дублируется. В первую копию добавляется условие перехода, во вторую — его отрицание. Если условие данного перехода было истинным на текущих входных данных, вторая копия набора условий сохраняется отдельно, как набор, инвертирующий этот переход. Первая же копия используется для дальнейшего построения набора.

После этого, либо после завершения выполнения программы, построенный набор ограничений должен быть проверен на выполнимость, что равносильно проверке существования входных данных, удовлетворяющих этим ограничениям.

2.3 Итерационный механизм

Первый запуск анализа происходит на начальных входных данных. Это могут быть корректные, частично-корректные или совершенно произвольные входные данные. В ходе первого запуска фиксируются все встретившиеся условные переходы и значения условных выражений в этих переходах. Последовательность таких значений представляет собой двоичный вектор. Для краткости будем использовать 0 в значении «ложь» и 1 в значении «истина».

Допустим, после запуска вектор переходов выглядит следующим образом: (1, 0, 1, 0) — первый условный переход выполнился, второй не выполнился и т. д.

По числу переходов строятся векторы, инвертирующие каждый из них:

- (0);
- (1, 1);
- (1, 0, 0) и
- (1, 0, 1, 1).

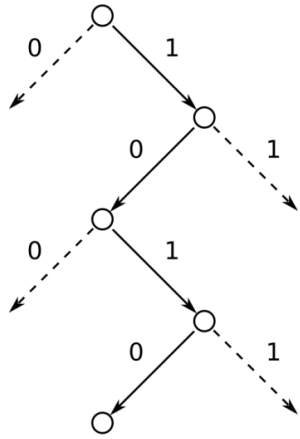


Рис. 2: Инвертирование переходов. Сплошной линией изображён ход выполнения приложения, пунктирными — возможные ветвления.

Каждый вектор соответствует очередному шагу работы анализа и набору утверждений, которые в случае выполнимости обеспечивают такие входные данные, на которых выполнимость первых n условных переходов (где n — длина вектора) соответствует значениям вектора.

Если рассматривать начальный вектор инвертированных переходов как вершину некоторого графа, а построенные векторы, как её дочерние вершины, то имеем упорядоченное корневое дерево. Потомки упорядочены по длине их векторов.

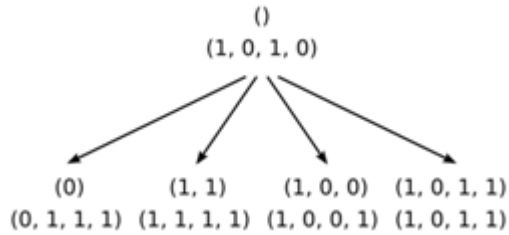


Рис. 3: Дерево переходов после первого шага. В вершинах изображены инвертирующий вектор и вектор запуска.

Для каждого из построенных векторов (т. е. для каждой построенной дочерней вершины) будет предпринята попытка генерации входных данных таким образом, чтобы в трассе запуска программы на этих данных первые n переходов соответствовали указанным в векторе инвертированных переходов.

После запуска вновь получаем вектор переходов. Если входные данные были построены верно, первые n значений в нём совпадают со значениями инвертирующего вектора.

Если длина нового вектора больше длины инвертирующего, все значения, начиная с $(n + 1)$ должны быть инвертированы. Если же длины векторов равны, вершина считается листовой и разбор данной ветки завершается.

Допустим, следующей будет раскрыта вершина с вектором (0).

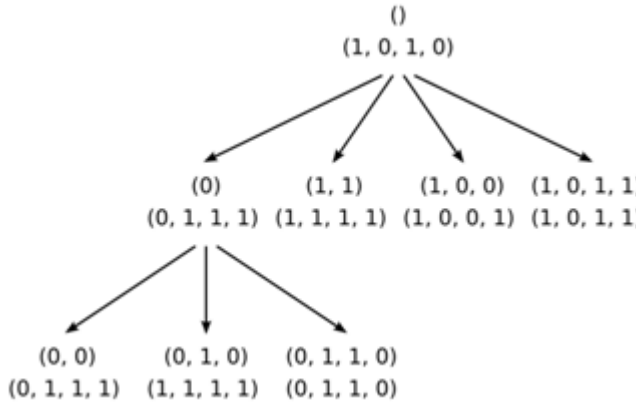


Рис. 4: Дерево переходов после второго шага.

На рисунке видно, что вершины с векторами (1, 0, 1, 1) и (0, 1, 1, 0) — листовые, а вершины (1, 1), (1, 0, 0), (0, 0) и (0, 1, 0) должны быть раскрыты.

Получаем дерево разбора всевозможных путей выполнения программы. Если в программе присутствуют m условных переходов, число вершин в дереве разбора может достигать 2^m. Соответственно, анализ всех путей выполнения программы может потребовать слишком большого времени. В связи с этим необходимо найти эвристики, которые позволят проводить целенаправленный поиск дефектов на наиболее вероятных или интересных с точки анализа путях исполнения программы.

2.3.1 Эвристики

После каждого очередного шага имеется набор векторов и соответствующие им утверждения. Необходимо иметь стратегию определения следующего вектора для анализа — наиболее перспективного с точки зрения обнаружения ошибок. Такая стратегия может быть выбрана, исходя из особенностей анализируемого приложения.

Наиболее простые стратегии — стратегии, не основывающиеся на какой-либо информации:

- обход дерева переходов в ширину,
- обход вглубь (с ограничением глубины или без).

Также можно предложить простые эвристические оценки для определения наиболее перспективной ветки — метрики трассы, полученной на предыдущем шаге:

- число инструкций,

- число потенциально опасных операций (например, целочисленное деление),
- число встретившихся условных переходов.

3. Прототип

Для проверки применимости предложенных методов к анализу программ на языке Java был разработан прототип инструмента динамического анализа. В этой части будут рассмотрены основные используемые средства и подходы.

3.1 Общая схема инструмента

В общих чертах схема инструмента может быть представлена следующим образом: шаг статической инструментации и далее вход в основной цикл работы, в ходе которого производится выполнение исследуемой программы, интерпретация и генерация входных данных.

Рис. 5: Общая схема инструмента

3.2 Инструментация

Описываемый прототип работает со стандартным байт-кодом, транслируемым, например, утилитой JDK javac и интерпретируемым, например, JVM (Java Virtual Machine).

Ограничив инструментацию определённым стандартом байт-кода, стоит отметить основное преимущество статической инструментации перед динамической. Динамическая инструментация происходит во время выполнения программы и может проводиться только при запуске программы на виртуальной машине, интерпретирующей соответствующий стандарт байт-кода. При этом, в случае статической инструментации, если существуют средства перевода между различными стандартами байт-кода,

инструментированный код может быть преобразован в необходимый формат без потери необходимой функциональности и анализ может быть произведён на виртуальной машине, не поддерживающей исходный формат байт-кода.

Примером использования этого преимущества может быть анализ программ на платформе Android, приложения на которой исполняются на виртуальной машине Dalvik и, таким образом, обязаны иметь иной формат байт-кода. Инструментированный байт-код может быть преобразован в формат DEX для анализа на целевой платформе.

Для проведения статической инструментации в прототипе используется библиотека BCEL — Byte Code Engineering Library [6]. Эта библиотека позволяет извлекать из класс-файлов информацию о содержащихся в них методах:

- сигнатуру метода,
- список констант, использующихся в методе,
- список инструкций и их параметры.

Также BCEL предоставляет возможности по изменению байт-кода: добавление, удаление, изменение инструкций; изменение списка констант; генерация новых методов.

На первом этапе работы инструмента происходит

- построение списка используемых в байт-коде методов,
- инструментация байт-кода исходного класс-файла и всех методов из списка.

В связи с тем, что обработка информации происходит в процессе выполнения программы, в прототип включены классы, предоставляющие необходимую функциональность по обработке информации о ходе выполнения программы. Инструментация заключается в добавлении в байт-код для каждой инструкции, если это требуется, предварительного и/или заключительного набора инструкций, добавляемых до и после этой инструкции соответственно. Каждый из этих наборов состоит из инструкций, производящих сохранение, копирование и минимальное преобразование различного рода аргументов, а также инструкций-вызовов методов классов, производящих обработку скопированной информации.

3.3 Генерация ограничений

Запуск анализируемой программы происходит с использованием инструментированных библиотек класс-файлов.

Для отслеживания помеченных данных и генерации ограничений на входные данные в процессе выполнения программы каждому значению ячейки памяти

(элемент стека или регистр) ставится в соответствие некоторая метапеременная. Изначально метапеременные сопоставляются байтам входных файлов. Если в ходе выполнения программы встречается операция, возвращающая значение, для каждого операнда проверяется, существует ли для него соответствующая метапеременная. Отсутствие метапеременной для всех аргументов означает, что в операции не участвуют помеченные переменные и она должна быть проигнорирована. Если же такие аргументы существуют, то создаётся новая метапеременная для результата операции и сопоставляется с ячейкой памяти, в которую результат записывается. В набор условий добавляется логическое утверждение — тождественное равенство, в левой части которого располагается созданная метапеременная, в правой части — метапеременные, соответствующие аргументам, или их извлечённые значения переменных, если метапеременные для них отсутствуют, связанные соответствующими операциями.

Для каждой операции условного перехода, как было сказано ранее, набор построенных ограничений дублируется, если в условие перехода входят переменные, которым сопоставлены метапеременные. Как и в случае с операциями, генерирующими значение, строится логическое выражение над метапеременными и извлечёнными значениями, соответствующее условию перехода в первой копии набора и соответствующее отрицанию этого условия во второй.

3.3.1 Проверка выполнимости ограничений

Для проверки выполнимости построенных ограничений используется инструмент STP — универсальный решатель логических ограничений [2].

На вход STP принимает последовательность логических утверждений — булевых операций над булевыми векторами — и как результат выдаёт сообщение о том, что данная последовательность является непротиворечивой, либо его ответом является контрпример — набор значений булевых векторов, при которых цепочка утверждений ложна.

Принцип работы инструмента STP заключается в переводе цепочки утверждений в конъюнктивную нормальную форму, после чего задача сводится к определению выполнимости булевой формулы, работа по проверки которой передаётся другому, более общему решателю [2].

В язык запросов STP специально разработан для поддержки основных логических операций, таких как логическое умножение и сложение, а также ряд арифметических операций, конструкция if — then — else и предикаты сравнения чисел, что делает его подходящим для решения поставленной задачи.

4. Результаты

4.1 Расход памяти при инструментации

В этом разделе приводится оценка увеличения размера класс-файлов после инструментации на примере набора пакетов.

Табл. 1: Изменение размеров файлов после инструментации

Пакет	Размер после инструментации	Размер до инструментации	Коэффициент
applet	20,2 КБ	7,3 КБ	2,7
awt	13,1 МБ	1,9 МБ	6,9
beans	1,9 МБ	359 КБ	5,3
io	2,3 МБ	352,2 КБ	6,5
lang	3,2 МБ	583 КБ	5,5
math	1,1 МБ	101,3 КБ	10,9
net	2,1 МБ	346,0 КБ	6,1
nio	2,1 МБ	376,5 КБ	5,6
rml	411,7 КБ	111,3 КБ	3,7
security	2,2 МБ	456,5 КБ	4,8
sql	283,7 КБ	102,4 КБ	2,8
text	2 МБ	279,6 КБ	7,2
util	11,1 МБ	1,9 МБ	5,8

В целом пакет увеличился в 6.16 раз. Похожий коэффициент можно ожидать от практически произвольного класс-файла.

4.2 Скорость работы

4.2.1 Инструментация

Скорость инструментации проверялась на классах из пакета rt.jar (реализация стандартной библиотеки классов Java для виртуальной машины HotSpot). Этот

архив содержит в себе 17 073 класс-файлов. Ниже приведены временные отметки инструментации этих классов:

Табл. 2: Время инструментации

Архив	Число классов	Время (секунды)	Скорость (классов в секунду)
rt.jar	16 796	96,220	174,55
tools.jar	3 679	21,623	170,14
ServeceabilityAgent	1 909	7,6	251,18
BCEL	383	2,568	149,14
JConsole	220	1,807	121,74
HTML Converter	52	1,353	38,43
dt.jar	47	0,878	53,53

Таким образом, за секунду инструментируется порядка сотни класс-файлов.

Такая скорость является приемлемой, поскольку инструментация производится один раз в ходе подготовки к итерации анализа и инструментируются не все подключённые класс-файлы, а те, чьи методы могут быть вызваны в ходе выполнения приложения.

Также на основе особенностей метода и приведённых результатов можно сделать вывод о том, что скорость инструментации линейно зависит от размера класс-файлов.

4.2.2 Выполнение

Проверка эффективности работы инструмента осуществлялась на приложениях с искусственно внесёнными ошибками (результаты приведены в таблице 6.3).

Тестовый пример. Демонстрирует основные возможности инструмента по нахождению ошибок, воспроизведение которых требует истинности большого числа условий в программе. Для такого приложения самым эффективным будет обход дерева разбора в глубину.

CSVReader. Представляет собой средство разбора простого формата данных CSV — набора слов и разделителей. Ошибка заключается в указании размера набора слов при помощи константного значения в коде программы. Инструмент обнаруживает входные данные, число слов в которых превышает

данную константу и вызывает исключение, связанное с выходом за границы массива. В этом случае наиболее эффективной оказывается эвристика, основывающаяся на подсчёте числа условных переходов.

Поиск пути в графе. Во входном файле задаётся граф в виде матрицы смежности и номера двух его вершин. Программа возвращает кратчайший путь из первой вершины во вторую. Ошибка заключается в том, что неверно оценивается размер буфера для хранения рёбер найденного пути.

Средство разбора арифметических выражений. Ошибка заключается в ограниченности размеров дерева разбора. Здесь также подсчёт числа условных переходов даёт наилучший результат.

Табл. 3: Время инструментации

Программа	Эвристика	Тип ошибки	Номер итерации и время обнаружения первой ошибки (сек.)
Тестовый пример	В глубину	Деление на ноль	4 (0,32 с)
	Число инструкций		7 (0,54 с)
CSVReader	В глубину	Выход за границы массива	43 (5,06 с)
	Число инструкций		39 (4,78 с)
	Число условных переходов		39 (4,78 с)
Поиск пути в графе	В глубину	Выход за границы массива	30 (6,51 с)
	В ширину		30 (6,5 с)
	Число условных переходов		32 (7,1 с)
Разбор арифметических выражений	В ширину	Выход за границы массива	29 (4,8 с)
	Число условных переходов		27 (4,6 с)

На основе приведённых результатов можно сделать вывод о том, что прототип инструмента демонстрирует эффективную работу по обнаружению ошибок, воспроизведение которых требует большого числа одновременно выполняющихся условий. При этом прототип инструмента не использует информацию о логике работы программы.

5. Заключение

В результате проведённого исследования был реализован прототип средства, осуществляющего динамический анализ приложений на языке Java.

В инструменте применена статическая инструментация байт-кода с целью получения трассы выполнения программы. В ходе анализа происходит извлечение трассы, частичная интерпретация и отслеживание потока помеченных данных, создание набора логических утверждений для инвертирования условных переходов и генерация новых входных данных для покрытия новых путей выполнения программы.

После проведения ряда тестов была подтверждена эффективность статической инструментации, проводимой один раз перед выполнением анализа.

В результате запуска анализа тестовых приложений инструмент продемонстрировал способность эффективно обнаруживать ошибки и создавать наборы входных данных для их воспроизведения без наличия исходного кода и информации о логике работы приложения. Генерация входных данных исключает возможность ложных сообщений об обнаруженных ошибках.

Для ускорения проведения анализа реализованы эвристические методы выбора наборов входных данных.

В качестве направлений для дальнейших исследований можно указать следующие:

- Разработка полноценного инструмента, позволяющего анализировать входные данные программ из различных источников: переменные окружения среды выполнения, сокеты и т. п.
- Разработка эвристик и алгоритмов для быстрого целенаправленного анализа интересующих частей приложения, например отдельно взятой функции или подсистемы в рамках приложения

Список литературы

[1]. Новикова Н. М. Основы оптимизации. М.: МГУ, 1998. 17–22 с.
[2]. Eén N., Sörensson N. MiniSat solver [HTML] (http://minisat.se/)
[3]. Ganesh V., Dill D. L. A Decision Procedure for Bit-Vectors and Arrays // In Proceedings of Computer Aided Verification. 2007. P. 524–536.
[4]. Исаев И. К., Сидоров Д. В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах // Программирование. 2010. № 4. С. 1-16.
[5]. Valgrind. Instrumentation Framework for Building Dynamic Analysis Tools [HTML] (http://valgrind.org/)
[6]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

Applying dynamic analysis for defect detection in Java-applications

S. Vartanov, A. Gerasimov
svartanov@ispras.ru, agerasimov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. This paper provides an overview of program analysis techniques, and highlights details of practical implementation of static and dynamic approaches for automatic software defect detection, their pros and cons. The paper focuses on dynamic program analysis technique. The major advantage of this technique is the absence of defect reproduction problem. This approach is based on tainted data flow tracing, instrumentation and constraint set construction for automatic input generation.

An overview of practical considerations for developing a dynamic analysis tool for Java applications is given. The paper describes distinctive feature of Java bytecode static instrumentation approach and related static dependencies detection problems. It provides details of path conditions generation (set of Boolean formulas), paths coverage-based iterative mechanism. Input generating problem is solved using solver for checking Boolean constraints satisfiability. Heuristics are used for exponential growth problem solving.

It is complemented by a detailed description of actual prototype implementation created within the scope of this project. The prototype uses BCEL for static instrumentation and STP solver for path condition solving. Finally, the paper features an overview of practical results obtained on a number of Java applications and provides an evaluation of these results.

Keywords: software iterative dynamic analysis; automatic defect detection; Java bytecode instrumentation

References

[1]. Novikova N. M. Osnovy optimizatsii [The Basics of Optimization]. M.: MGU [MSU], 1998. 17–22 p. (in Russian)
[2]. Eén N., Sörensson N. MiniSat solver [HTML] (http://minisat.se/)
[3]. Ganesh V., Dill D. L. A Decision Procedure for Bit-Vectors and Arrays. In Proceedings of Computer Aided Verification. 2007. P. 524–536.
[4]. Isaev I. K., Sidorov D. V. Primenenie dinamicheskogo analiza dlya generatsii vkhodnykh dannykh, demonstriruyushhikh kriticheskie oshibki i uyazvimosti v programmakh [The Use of Dynamic Analysis for Generation of Input Data that Demonstrates Critical Bugs and Vulnerabilities in Programs]. Programmirovanie [Programming and Computer Software]. 2010. # 4. P. 1-16. (in Russian)
[5]. Valgrind. Instrumentation Framework for Building Dynamic Analysis Tools [HTML] (http://valgrind.org/)
[6]. Apache Commons Byte Code Engineering Library [HTML] (http://commons.apache.org/bcel)

Avalanche: применение параллельного и распределенного динамического анализа программ для ускорения поиска дефектов и уязвимостей

М. К. Ермаков, А. Ю. Герасимов[1]
mermakov@ispras.ru, agerasimov@ispras.ru

Аннотация. В статье рассматривается подход к уменьшению времени динамического анализа программ при помощи применения параллельных вычислений при проверке выполнимости ограничений, а также при применении распределенных вычислений в процессе динамического анализа программ. Приводятся результаты практического применения данного подхода, реализованного в рамках инструмента динамического анализа Avalanche. На основе полученных результатов даётся оценка увеличения эффективности динамического анализа и рассматриваются возможности дальнейшего развития разработанных методов.

Ключевые слова: динамический анализ, обнаружение дефектов, параллельные вычисления, распределенные вычисления.

1. Введение

Решение задачи поиска дефектов и уязвимостей в программах при помощи динамического анализа программ инструментом Avalanche [1] сводится к решению следующих подзадач:

- Отслеживание потока помеченных данных в анализированной программе с параллельным сбором информации об условных переходах на трассе анализа программы при помощи дополнительного модуля Tracegrind для среды динамической инструментации программ Valgrind [2].
- Вычисление новых входных данных для программы с целью воспроизведения дефекта в потенциально опасной операции и обхода ещё не пройденных ветвей в программе при помощи решателя STP [3].

1 Работа проводится в рамках научно исследовательских работ Института системного программирования РАН в 2012-2013 годах

- Выбор наиболее предпочтительного набора входных данных для следующей итерации анализа приложения

Изначально, в инструменте Avalanche применялся последовательный подход при анализе программы (рис. 1).

1. Запускался анализ на некотором начальном наборе входных данных, собиралась трасса выполнения программы, на которой запоминались данные о потенциально опасных операциях и условных переходах, зависящих от входных данных.
2. При помощи решателя STP последовательно вычислялись новые наборы входных данных для
    - каждой потенциально опасной операции с целью генерации входных данных, подтверждающих наличие дефекта в программе
    - каждого условного перехода, зависящего от входных данных, с целью обхода не пройденных частей программы
3. Производился выбор наиболее перспективного набора входных данных для следующей итерации анализа на основе метрики наибольшего прироста не проанализированных базовых блоков программы.

Рис. 1: Общая схема работы инструмента Avalanche

В связи с тем, что инструментация и выполнение инструментированной программы средой Valgrind является достаточно тяжеловесной операцией, а также вычисление выполнимости формулы решателем STP в общем случае является NP-полной задачей, оптимизация вычисления которой в случае применения вычислителя при проведении динамического анализа программ описана в работе Вартанова и

Сидорова [4], то проведение динамического анализа программы занимает существенное время. В связи с этим встает задача оптимизации работы инструмента Avalanche для ускорения процесса анализа программы. В данной статье будут рассмотрены подходы к распараллеливанию вычислений и реализация распределенных вычислений в рамках инструмента Avalanche.

2. Параллельный динамический анализ

При анализе возможных вариантов решения задачи распараллеливания работы инструмента Avalanche рассматривались варианты распараллеливания работы дополнительного модуля Tracegrind для среды Valgrind и распараллеливания вычисления новых входных данных для следующих запусков анализа программы.

Реализация распараллеливания работы дополнительного модуля Tracegrind оказалась невозможна в связи с ограничениями инструментирующей среды Valgrind. В связи с этим предложен подход к распараллеливанию вычисления новых входных данных для следующей итерации динамического анализа программы. На рис. 2 показана принципиальная схема распараллеливания вычислений.

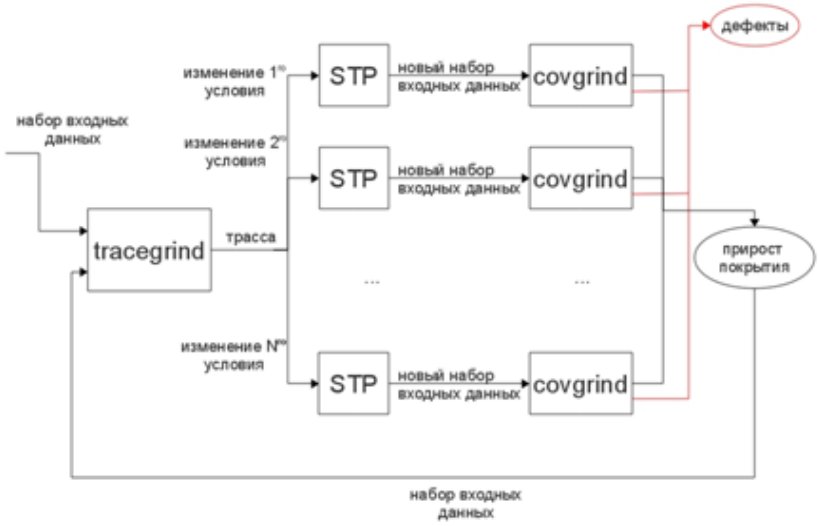


Рис. 2: Схема проведения параллельного анализа

В процессе работы дополнительного модуля Tracegrind для среды Valgrind собирается информация о трассе выполнения программы, а именно о потенциально опасных операциях и условных переходах, зависящих от входных данных. Количество условных переходов и

потенциально опасных операций на пути выполнения программы может достигать достаточно большого количества.

Для каждой потенциально опасной операции, найденной на трассе выполнения программы, необходимо произвести расчет булевской формулы при помощи STP, которая может подтвердить или не подтвердить возможность воспроизведения ошибки при выполнении потенциально опасной операции. В то же время для каждого условного перехода, зависящего от входных данных программы, необходимо вычислить булевскую формулу при помощи SAT, которая позволит определить входные данные для обхода альтернативной ветки условного перехода.

Исходя из данных, приведенных в работе [1], время работы вычислителя может занимать более 99% общего времени работы инструмента для некоторых проектов, а вычисление булевской формулы может производиться в от дельном вычислительном потоке, то, выполняя вычисление булевских формул параллельно, можно значительно сократить общее время анализа программы.

В табл. 1 приведены данные экспериментальных запусков инструмента Avalanche на наборе тестовых приложений с включенным и выключенным режимом распараллеливания работы STP. Для всех указанных проектов были использованы одинаковые начальные параметры, время работы было ограничено 2 часами (7200 секундами).

Табл. 1. Сравнительные результаты работы инструмента Avalanche в однопоточном и параллельном режимах

	Однопоточный режим анализа		Параллельный режим анализа		
Программа	Количество итераций	Количество дефектов	Количество итераций	Количество дефектов	Ускорение
qtdump	142	2	68	2	-115%
cjpeg	62	1	109	1	+76%
sndfile-mix-to-mono	48	0	313	1	+552%
avibench	64	2	171	1	+167%
mpeg2dec	200	0	661	1	+230%
mpeg3dump	151	2	283	2	+87%

Применение алгоритма определения лучшего прироста покрытия анализируемой программы, работающего при параллельном режиме в отдельном потоке на отдельной булевой формуле, вводит элемент неопределенности при выборе следующего наилучшего набора входных данных программы. Если вычисление каждой из нескольких формул занимает примерно одинаковое время, то существует вероятность получения лучшего набора данных для увеличения покрытия анализируемой программы для разных булевых формул в зависимости от работы планировщика потоков вычислительной среды, что вводит некоторую неопределенность при оценке времени нахождения определенного дефекта в анализируемой программе. Решить эту проблему можно введением некоторого арбитра, который будет накапливать набор данных для дальнейшего анализа и выбирать лучший из нескольких полученных.

На практике данная особенность может приводить к различного рода последствиям, как положительным с точки зрения качества анализа, так и отрицательным. В частности, выбор наилучшей ветки для проведения следующей итерации, отличающейся от ветки, выбранной при использовании последовательной схемы анализа, может как позволить определить ранее не обнаруженный дефект, так и не провести проверку определённой ветки выполнения с потерей ошибки, воспроизводимой на данной ветке.

Так, например, на проекте avibench, несмотря на увеличения количества итераций анализа, был не обнаружен дефект, связанный с разыменованием нулевого указателя, воспроизводящимся на единственной ветке выполнения. В другом случае, на проекте qtdump детерминизм, вносимый параллельными вычислениями, вызвал более быстрое увеличение “глубины” просмотра точек ветвления программы. Подобное “углубление” привело к росту сложности трасс и, соответственно, запросов для решателя STP. В конечном итоге большая часть времени анализа была затрачена на проверку выполнимости полученных трасс, что снизило количество итераций анализа. Падение производительности было зафиксировано только на проекте qtdump и наиболее вероятно связано с особенностями трасс, получаемых при выполнении программы.

3. Распределенный динамический анализ

В работе Исаева и Сидорова [1] выполнение каждой итерации анализа моделируется как вычисление на одной из веток дерева достижимых путей в программе. И делается предположение, что используя данную модель можно добавить возможность запуска инструмента Avalanche на некотором поддереве достижимых путей в программе, используя одно из ветвлений на пути выполнения программы как корень поддерева.

Применение подобной возможности позволит выполнять запуск инструмента Avalanche на нескольких компьютерах или в распределенной вычислительной среде, используя её вычислительные возможности.

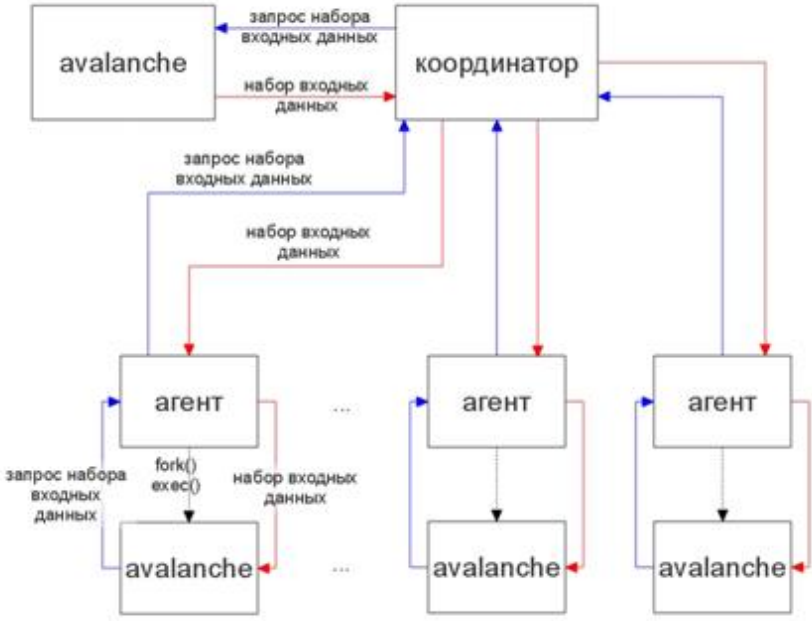


Рис. 3: Схема работы инструмента Avalanche при использовании распределённых вычислительных ресурсов

Для реализации этой возможности модифицирован алгоритм работы инструмента Avalanche:

1. В качестве входного параметра работы инструмента на ряду с набором входных данных анализируемой программы передается информация о глубине условного перехода, начиная с которой должно производиться инвертирование условия условного перехода для вычисления входных данных с целью обхода альтернативного пути выполнения программы.
2. Реализован координатор, который накапливает вычисленные наборы входных данных анализируемой программы и имеет возможность передачи набора входных данных и глубины условий для инвертирования на свободный узел вычислительной среды.

3. Реализован промежуточный агент, которая запускается на узле вычислительной среды и:
    - ожидает получения данных для анализа целевой программы от координатора
    - запускает инструмент Avalanche для анализа поддерева возможных путей программы.

Каждый агент по сути превращает узел вычислительной среды в независимый анализатор программы, который позволяет производить полноценный анализ подмножества возможных путей выполнения программы, вычисляя входные данные для обхода новых путей выполнения в программе и подтверждения дефекта для потенциально опасной операции. При этом подмножества путей, анализируемые на различных вычислительных узлах, не пересекаются между собой, за исключением редких случаев недетерминизма внутри самой исследуемой программы.

Особенность текущей реализации распределенного режима инструмента Avalanche позволяет агентам получать данные для анализа только от координатора. Координатор работает на одном из узлов распределенной вычислительной среды и накапливает наборы входных данных для анализа альтернативных путей в программе. Особенность текущей реализации координатора заключается в том, что когда на узле, где работает анализатор, проанализированы все достижимые пути на тех поддеревьях, которые не переданы для анализа на агенты, то анализ на узле координатора заканчивается. Также, данные о дефектах, найденных на агенте накапливаются на узле, где был запущен агент, что при использований общего сетевого диска не является серьезным ограничением.

Результаты работы инструмента Avalanche в распределенном режиме представлены в табл. 2. Данные результаты были получены при использовании сети из 4 вычислительных узлов со сходными характеристиками, время работы анализа было ограничено 2 часами (7200 секундами). Для сравнения эффективности анализа используются два критерия: количество обнаруженных дефектов и время обнаружения данных дефектов, рассчитываемое с момента старта анализа (для распределённого режима приводится наименьшее время обнаружения соответствующего дефекта среди всех вычислительных узлов).

Табл. 2. Сравнительные результаты работы инструмента Avalanche в обычном и распределённом режиме

	Стандартный режим анализа		Распределённый режим анализа	
	Время обнаружения дефектов, с	Количество дефектов	Время обнаружения дефектов, с	Количество дефектов
qtdump	2872 3110	2	137 6810 7048	3
cjpeg	1114	1	330 2160	2
mpeg2dec	-	0	410 5128	2
mpeg3dump	1 706	2	1 103 280	3

Результаты применения инструмента в распределённом режиме свидетельствуют о достаточной эффективности использования данного режима при наличии соответствующей вычислительной базы. Разделение проверки поддеревьев общего дерева возможных путей выполнения приводит к тому, что за ограниченное время происходит обнаружение большего числа дефектов. Также происходит уменьшение времени, необходимого для обнаружения дефектов, по сравнению с использованием только одного вычислительного узла.

Распределённый анализ, подобно паралелльному анализу, может вносить достаточную степень недетерминизма в процесс выбора наборов входных данных для последующих итераций. При использовании только одного вычислительного узла информация о приросте покрытия ещё не рассмотренных путей выполнения программы применяется на всех итерациях анализа. При использовании нескольких вычислительных узлов информация о приросте покрытия является локальной для каждого вычислительного узла, что может изменить непосредственные значения прироста для различных путей выполнения (по сравнению со значениями, которые бы наблюдались при стандартном режиме анализа). Так, как и при использовании параллельного анализа, недетерминизм привёл к увеличении времени обнаружения дефектов в проекте qtdump, однако позволил обнаружить дополнительную уязвимость.

Стоит отметить, что работа в распределенном режиме может быть улучшена путем применения распределенного алгоритма работы

координатора, который бы позволил эффективно обрабатывать ситуацию, когда на вычислительном узле координатора у анализатора Avalanche заканчиваются данные о достижимых, но не проанализированных путях в анализируемой программе. Это позволит осуществлять более полный анализ целевой программы за ограниченное время. В этом случае одним из вариантов продолжения анализа программы может стать получение достижимых, но ещё не проанализированных путей от агентов и передача входных данных на другого агента, который закончил анализ переданного ему поддерева. Снижение детерминизма, вносимого применением распределённых вычислений, также является одним из перспективных направлений дальнейших исследований.

4. Заключение

В статье рассмотрены способы улучшения производительности динамического анализа программ при помощи параллельного и распределенного анализа на примере решений реализованных в инструменте динамического анализа программ Avalanche. Описаны схемы реализации параллельного и распределенного режимов работы анализатора, ограничения инструмента и технологий, на которых построен инструмент, при применении рассмотренных методов и приведены результаты подтверждающие обоснованность предложенных решений, а также указаны направления дальнейших исследований.

Список литературы.

[1]. И. К. Исаев, Д. В. Сидоров. «Применение динамического анализа при генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах». Программирование, №4, 2010, с. 1-16 .

[2]. Nethercote N., Seward J. “Valgrind: A framework for heavyweight dynamic binary instrumentation”. PLDI, 2007.

[3]. V. Ganesh and D. Dill. “A decision procedure for bit-vectors and arrays”. In CAV 2007, LNCS 4590, pages 519-531

[4]. С.П. Вартанов, Д. В. Сидоров «Оптимизация задачи проверки выполнимости булевских ограничений при помощи кэширования промежуточных результатов». Труды института системного программирования РАН, том 22, 2012, с. 281-292.

Avalanche: adaptation of parallel and distributed computing for dynamic analysis to improve performance of defect detection

M. K. Ermakov, A. Y. Gerasimov
mermakov@ispras.ru, agerasimov@ispras.ru
ISP RAS, Moscow, Russia

Abstract. This paper focuses on dynamic program analysis optimization through the use of distributed computing scheme and parallel computing for checking satisfiability of Boolean constraint sets. The paper is organised as follows: section 1 contains an overview of the Avalanche tool and identifies the key points of its work flow and module structure applicable to distributed and parallel optimizations. Sections 2 and 3 provide general schemes of parallel and distributed program analysis respectively: parallel approach is based on the relative independence of several execution constraint sets; distributed approach is based on a protocol for communication between a number of computing units targeted at sharing sub-trees of execution paths in the analysed program. An overview of results obtained from applying the practical implementation of parallel and distributed schemes of dynamic analysis to a number of open-source applications is given. The paper presents a detailed evaluation of the increased efficiency (in terms of the number of defects detected, as well as relative detection time) of dynamic analysis achieved while applying developed techniques. Section 4 concludes the paper with an overview of possible issues related to non-deterministic nature of parallel and distributed approaches and discusses future directions for research on the Avalanche tool.

Keywords: dynamic analysis, defect detection, parallel computing, distributed computing.

References

[1]. I. K. Isaev, D. V. Sidorov. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. Programming and Computing Software. Volume 36 Issue 4, July 2010. pp. 225-236. doi:10.1134/S0361768810040055

[2]. Nethercote N., Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. Proceedings of ACM SIGPLAN conference on Programming language design and implementation, 2007. pp. 89-100. doi: 10.1145/1250734.1250746

[3]. V. Ganesh and D. Dill. A decision procedure for bit-vectors and arrays. Proceedings of the 19th international conference on Computer aided verification, 2007. pp. 519-531

[4]. S. P. Vartanov, D. V. Sidorov. Optimizatsiya zadachi proverki vypolnimosti bulevskikh ogranichenij pri pomoshhi kehshirovaniya promezhutochnykh rezul'tatov. [Optimization of Boolean satisfiability solver by caching intermediate results]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 281-292 (in Russian).

Поиск повторяющихся фрагментов исходного кода при автоматическом рефакторинге

Н. Г. Зельцер
nzeltser@ispras.ru

Аннотация. В статье рассмотрена возможность совмещения автоматического рефакторинга с обнаружением повторяющихся фрагментов исходного кода для программ на языках C/C++. Предложена классификация программных клонов с точки зрения дальнейшего применения к ним автоматического рефакторинга. Для каждого выделенного типа клонов описан способ их поиска. Приведены недостатки существующих инструментов и показано, что предложенные методы работают корректно в рассмотренных ситуациях. Подход, описанный в статье, реализован в рамках инструмента Klocwork inSight.

Ключевые слова: рефакторинг, дубликаты кода.

1. Введение

Наличие повторяющихся фрагментов исходного кода влечет несколько проблем:

- неоправданное увеличение количества строк, а значит, ухудшение читаемости программы;
- ухудшение дизайна программы;
- усложнение поддержки программной системы;
- повышение затрат на исправление ошибок в программе, если ошибка найдена в повторяющемся коде, поскольку необходимо вносить изменения во все копии повторяющегося кода.

Для улучшения дизайна программы большую популярность приобрел рефакторинг “Выделение метода” [1](с. 110-116). Если выделенный пользователем фрагмент кода имеет дубликаты, то стоит попробовать применить тот же рефакторинг также и к ним. Таким образом, совмещение рефакторинга “Выделение метода” с поиском клонов позволяет существенно улучшить структуру программы.

Компилятор Klocwork inSight для языков C и C++ на одном из этапов работы представляет исходный код программы в виде дерева разбора на

базе дерева абстрактного синтаксиса (AST) с необходимой семантической информацией для каждого узла. Учитывая возможности компилятора по сохранению информации об оригинальной разметке исходного кода (индентации, макросах и комментариях), то возникает возможность осуществить поиск клонов с одной стороны наиболее точно и, в то же время, независимо от индентации, макросов и комментариев, сосредоточившись только на структуре кода в виде дерева разбора.

2. Существующие решения

Как правило, большинство существующих инструментов поиска клонов для программ на языках C/C++ обладают рядом недостатков и дают неточные или мало полезные с точки зрения дальнейшего рефакторинга результаты. Это происходит потому, что поиск зачастую осуществляется на уровне исходного текста программы, практически без учета структуры и семантики кода. Не учитываются типы переменных, функций и их аргументов. Нет корректной поддержки кода, содержащего макро-вызовы.

В качестве примеров, приведем существующие инструменты CloneDR [2] и CCFinder [3]. Инструмент CloneDR позволяет находить группы повторяющихся фрагментов кода в данной пользователем директории проекта, не задавая какой-либо конкретный участок кода. Как было сказано выше, к результатам работы, вообще говоря, не применим один и тот же рефакторинг “Выделение метода”. Например, следующие два фрагмента будут выданы инструментом CloneDR в качестве клонов:

```
int i; double d;
...
if (flags[i].compare(“string1”) == 0) {  // Фрагмент 1
    if (i + 1 < flags.size()) {
      //do work ...
    }
}
...
if (flags[i].compare(“string2”) == 0) { // Фрагмент 2
    if (i + 1 < flags.size()) {
      //do work ...
    }
}
...
```

```
if (flags[i].compare(“string3”) == 0) { // Фрагмент 3
    if (d + 1 < flags.size()) {
      //do work ...
    }
}
```

CloneDR выдает параметризованный шаблон, которому соответствует найденная группа клонов. При этом, если пользователь желает выполнить рефакторинг “Выделение метода” для найденных фрагментов, выбор списка параметров остается за программистом: CloneDR ничего об этом не сообщает. В данном случае шаблон выглядит так:

```
if (flags[i].compare(“$variable1”) == 0) {
    if ($variable2 + 1 < flags.size()) {
      //do work ...
    }
}
```

Очевидно, что было бы неверным заменить фрагменты 1 и 3 на вызов одной и той же функции, поскольку переменные i и d - разных типов. В данном случае клонами являются только фрагменты 1 и 2, и для них стоит выделить следующую функцию:

```
bool extracted_function(const vector<string>, const char
*e, int i) {
    if (flags[i].compare(e) == 0) {
        if (i + 1 < flags.size()) {
            //do work
        }
        return true;
    }
    return false;
}
```

Подход к поиску программных клонов, реализованный в инструменте Klocwork, учитывает вышеописанные факторы и в результате приводит к верной замене повторяющихся фрагментов на вызовы выделенной функции.

Для языка Java неплохим решением является клон-детектор, реализованный как часть среды разработки Intellij IDEA. Пользователь выделяет фрагмент кода для рефакторинга, и для него автоматически

ищутся клоны. Здесь учитывается семантика кода: типы переменных, сигнатуры функций т.д. Также плюсом является то, что фрагменты кода, отличающиеся лишь некоторыми входящими в них арифметическими выражениями, выдаются как клоны, что придает гибкость процессу рефакторинга.

Однако, данный инструмент также имеет серьезные недостатки. Во-первых, отсутствует возможность поиска клонов вне текущей функции. Во-вторых, фрагменты кода, отличающиеся лишь входящими в них числовыми и строковыми константами, не обнаруживаются как клоны. Клон-детектор, реализованный в проекте Klocwork, описанных недостатков не имеет.

Не стоит также забывать, что анализ программ на языке Java с точки зрения поиска клонов и рефакторинга в общем случае проще, чем аналогичные действия для программ на языках C/C++ в силу присутствия в последних таких понятий, как, например, указатели и макросы.

3. Типы программных клонов

Итак, с точки зрения дальнейшего рефакторинга целесообразно выделить следующие типы клонов.

3.1 Точные клоны

Два фрагмента исходного кода называются точными клонами, если они синтаксически и семантически совпадают с точностью до имен идентификаторов.

Ниже приводится пример таких клонов.

Заданный фрагмент:

```
while(1){
int fd = 4;
    int e = a + b;
    printf("%f:%f\n", a, b);
    a++; b++;
    /* комментарий*/
    send_data(fd, &e);
    d = (double*)receive_data(fd);
    sum += *d *(e);
}
```

Точный клон:

```
while(1){
    int fd = 4;
    int e = x + y;
    printf("%f:%f\n", x, y);
    x++;
    y++;
    send_data(fd, &e);
    d = (double*)receive_data(fd);
    sum += *d *(e);
}
```

Важно отметить, что точные клоны могут отличаться форматированием текста, наличием или отсутствием комментариев. Очевидно, что для данных фрагментов исходного кода можно выделить метод с тремя параметрами.

В качестве тривиального подтипа можно выделить *идентичные* клоны: точные клоны, в которых также совпадают имена соответствующих идентификаторов. Этот подтип клонов полезен при рефакторинге “Введение переменной” [1](с. 124-127). Например:

```
if (b*b + 2*d + 3 > 0) { // Пользователь выделяет условное выражение
   a = b*b + 2*d + 3;  // Идентичный клон
}
w = q*q + 2*r + 3;      // Не идентичный клон
```

После проведения рефакторинга код будет выглядеть следующим образом:

```
int new_variable = b*b + 2*d + 3;
if (new_variable > 0) {
    a = new_variable;
}
w = q*q + 2*r + 3;
```

Ясно, что второй клон не стоит заменять на `new_variable`, в связи с использованием отличающихся переменных

Кроме того, интерес могут представлять фрагменты, идентичные по структуре и семантике, но отличающиеся используемыми в них константами. Иными словами, это фрагменты, являющиеся точными

клонами с точностью до констант. Отнесем им также к этому типу клонов.

3.2 Клоны 2-го типа

Фрагмент исходного кода называется клоном 2-го типа заданного фрагмента кода, если они совпадают с точностью до того, что для каждой переменной типа T, входящей в заданный фрагмент, на соответствующем месте в фрагменте-клоне находится либо переменная типа T, либо произвольное выражение типа T.

Здесь можно провести аналогию между переменной и wildcard-символом в шаблоне, на месте которого может стоять произвольное сложное выражение.

Пример:

Заданный фрагмент:

```
while (1) {
    int e = a; // переменная-wildcard
    printf("data = %d, counter = %d", e, c);
    c++;
    send_data(fd, &e);
    d = (double*)receive_data(fd); // fd - переменная
    sum += *d *(e);
}
```

Клон 2-го типа:

```
while (1) {
  int e = x + (x + 2)/bar(q); // сложное выражение на
месте переменной
  printf("data = %d, counter = %d", e, r);
  r++;
  send_data(fd, &e);
  z = (double*)receive_data(fd + 4); // сложное выражение
на месте переменной
  sum += *z *(e);
}
```

Этот тип программных клонов представляет интерес с точки зрения рефакторинга по той причине, что для таких фрагментов можно выделить один и тот же метод всего лишь четырьмя параметрами:

```
void extracted_function(double a, int *c, int fd, double
*sum){
  double *d;
  while(1){
    int e = a;
    printf("data = %d, counter = %d", e, *c);
    (*c)++;
    send_data(fd, &e);
    d = (double*)receive_data(fd);
    *sum += *d *(e);
  }
}
```

Тогда заданный фрагмент кода следует заменить на вызов выделенного метода со следующими фактическими параметрами:

```
extracted_function(a, &c, fd, &sum);
```

Фрагмент-клон необходимо заменить такой строчкой:

```
extracted_function(x + (x + 2)/bar(q), &r, fd + 4, &sum);
```

3.3 Клоны 3-го типа

Если в заданном фрагменте кода наряду с переменными рассматривать так же сложные выражения как wildcard-символы, то можно ввести еще один тип клонов.

Фрагмент исходного кода называется клоном 3-го типа заданного фрагмента кода, если они совпадают с точностью до того, что для каждой переменной v и каждого выражения expr типа T, входящего в заданный фрагмент, на соответствующем месте в клоне находится либо переменная типа T, либо произвольное выражение типа T.

Иными словами, клон 3-го типа - это такой фрагмент кода, который отличается от оригинала только тем, что на местах переменных или некоторых выражений в нем могут стоять переменные либо произвольные выражения того же типа.

Пример:

Заданный фрагмент:

```
while(1){
    int e = a — 28 * foo(1.0); //выражение - wildcard
    printf("data = %d, counter = %d", e, c);
    c++;
```

```
    send_data(fd, &e);
    d = (double*)receive_data(fd + 4); //выражение -
wildcard
    sum += *d *(e);
}
```

Клон 3-го типа:

```
while(1){
    int e = (x + 2)/bar(q - t) + x; //выражение на месте
wildcard-выражения
    printf("data = %d, counter = %d", e, r);
    r++;
    send_data(fd, &e);
    z = (double*)receive_data(fd); //переменная на месте
wildcard-выражения
    sum += *z *(e);
}
```

Для приведенных выше фрагментов исходного кода естественно выделить тот же метод, что в примере с клонами 2-го типа. Тогда вызовы выделенного метода будут выглядеть, как:

```
extracted_function(a – 28 * foo(1.0), &c, fd + 4, &sum);
extracted_function((x + 2)/bar(q - t) + x, &r, fd, &sum);
```

4. Подход для поиска программных клонов, реализованный в Klocwork Insight.

В инструменте Klocwork Insight реализована возможность автоматического рефакторинга заданного пользователем фрагмента кода. Среди поддержанных рефакторингов такие, как “Выделение метода”, “Введение переменной” [1] (с. 124-127).

4.1 Представление AST

На вход модулю поиска повторяющихся фрагментов кода поступает AST программы, AST выделенного фрагмента кода (либо AST ближайшего объемлющего узла, если выделена часть выражения) и тип клонов: точные или идентичные. На выходе модуля - список клонов с их границами в исходном коде, а также некоторая вспомогательная информация. Также поддержан поиск клонов 2-го и 3-го типов, однако на данный момент для них не проведена интеграция с модулем

автоматического рефакторинга. На рисунке ниже схематически изображена работа инструмента.

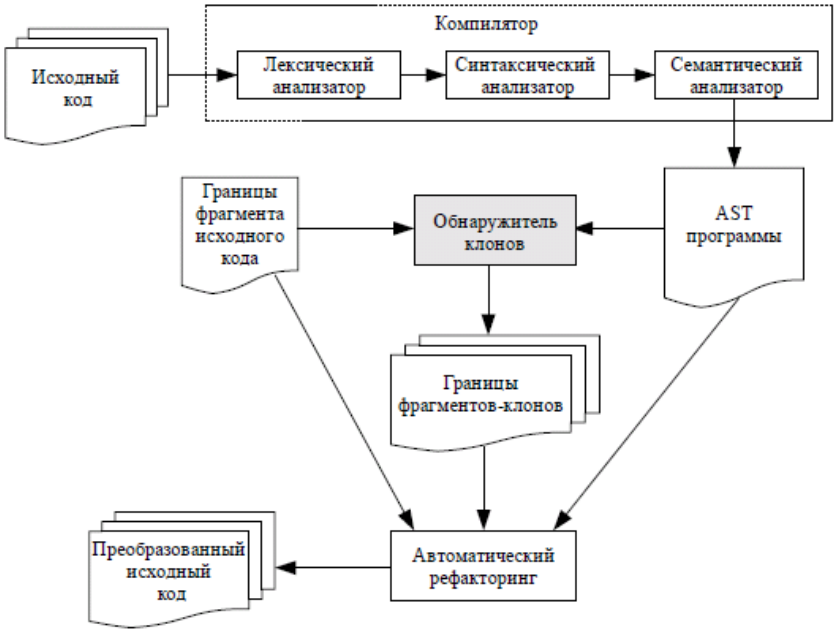


Рис. 1. Схема работы рефакторинга с поиском клонов в инструменте Klocwork

С целью поиска программных клонов, AST представляется в виде последовательности специально введенных *символов*. Для этого каждому типу узла сопоставляется числовой код и некоторая дополнительная информация, вид которой частично зависит от типа узла. Эта информация получается из семантической информации, хранимой в узле AST. Для подавляющего большинства типов узлов AST никакой дополнительной информации, кроме позиции узла в исходном коде, хранить не нужно. Интерес представляют лишь типы узлов, относящиеся к переменным и выражениям. Так, например, для узлов типа “Выражение” нужно хранить тип выражения; для узлов типа “Строковый литерал” - значение этого литерала; для узлов типа “Приведение типа” - тип исходного выражения и целевой тип. Всего для языка C++ можно выделить около 250-ти типов узлов AST.

4.2 Алгоритм поиска точных клонов

Задачу поиска точных программных клонов можно свести к задаче поиска подстроки в строке [4]. Алгоритм поиска таков:

1) Перевести AST выделенного фрагмента в последовательность символов;

2) Выполнить аналогичное действие для AST всей программы;
3) Найти во второй последовательности вхождения первой.

При этом нужно всего лишь задать отношение равенства для введенных нами выше *символов*. Например, если сравниваются два символа с одинаковым кодом, обозначающим узел-переменную, то нужно сравнить типы переменных.

4.3 Алгоритм поиска клонов 2-го типа

Поиск клонов второго типа можно проводить схожим способом. Немного меняется функция сравнения символов. В частности, необходимо сопоставлять символ, отвечающий узлу-переменной не только с символом с тем же кодом, но и с узлом, отвечающим узлу-выражению. При этом также обязательна проверка на совместимость типов переменной и выражения.

На рисунке изображен пример сопоставления строки-шаблона (сверху) с подстрокой основной строки (снизу). Числами обозначены коды символов. Символ с кодом 5 отвечает узлу AST - переменной, а символ с кодом 9 - узлу AST типа “Сложение”. Здесь мы сопоставили одному символу несколько символов - 9, 1, 6, 2, 8 - те, что отвечают поддереву AST с корнем-узлом типа “Сложение”.

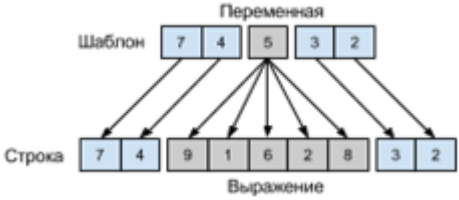


Рис. 2. Схема поиска клонов 2-го типа.

4.4 Алгоритм поиска клонов 3-го типа

Способ обнаружения таких клонов отличен от предыдущего, тем что в функцию сравнения символов добавляется симметричность. А именно: при обработке символа из строки-шаблона, отвечающего узлу AST-выражению стоит сопоставлять их не только с символом с тем же кодом, но и с с символом, отвечающим узлу AST-переменной. Также обязательна проверка на соответствие типов данных выражений. На рисунке обозначен пример соответствия строки-шаблона (сверху) с подстрокой строки, отвечающей всей программе.

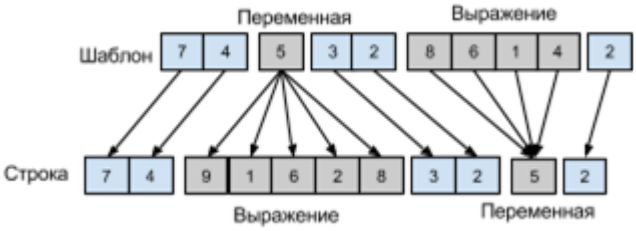


Рис. 3. Схема поиска клонов 3-го типа.

5. Примеры роботы обнаружителя программных клонов

Пример иллюстрирует работы инструмента в случае точных клонов, отличающихся лишь константами. Допустим, выделен фрагмент кода:

```
if(record.size() < 2) throw SyntaxError;
if(record.size() < 3) throw SyntaxError;
```

Тогда следующие фрагменты будут обнаружены как клоны:

```
if(record.size() < 10) throw SyntaxError;
if(record.size() < 5) throw SyntaxError;
…
if(record.size() < 3) throw SyntaxError;
if(record.size() < 4) throw SyntaxError;
```

В результате будет выделена новая фукнция `check_errors`, а приведенные выше участки кода - заменены на:

```
check_errors(2, 3);
check_errors(10, 5);
check_errors(3, 4);
```

Заключение

В статье предложен способ обнаружения повторяющегося кода с целью применения автоматического рефакторинга к найденным фрагментам. Выделено несколько типов клонов и предложены алгоритмы их поиска. Эффективность алгоритма подтверждена реализацией в инструменте Klocwork inSight. С точки зрения перспективных направлений развития данной темы можно выделить следующие.

1. Интеграция с модулем автоматического рефакторинга для случая клонов 2-го и 3-го типа.
2. Применение анализа потока данных в коде с целью более гибкого поиска клонов. Например, фрагменты могут отличаться порядком следования некоторых строк, но их перестановка не меняет функциональность программы. Кроме того, можно искать клоны, пренебрегая недосягаемым кодом или кодом, который не влияет на поведение программы.
3. Применение клон-детектора для других видов рефакторинга. Например, для рефакторинга “Замена условного оператора полиморфизмом”.

Список литературы:

[1]. M. Fowler., K. Beck, J. Brant, W. Opdyke, D. Roberts. Refactoring. Improve the design of exesting code. Addison-Wesley, 2001

[2]. http://www.semdesigns.com/products/clone/CCloneDR.html

[3]. http://www.ccfinder.net

[4]. Т.Кормен, Ч. Лейзерсон, Р.Риверст, К.Штайн Алгоритмы: построение и анализ.-М.: Вильямс, 2005.-1296 с.

Automatic clone detection for refactoring.

N. G. Zeltser
nzeltser@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Software clones are repeating fragments of source code. Clones are considered harmful for many reasons and should be eliminated from the program. In this paper we study the possibility to combine automatic refactoring with the detection of repeating fragments in C/C++ source code. Classification of software clones is proposed in terms of their further use during automatic refactoring. Three types of clones are defined depending on the level of rigor: exact clones, clones withing the accuracy of concrete arithmetical expressions and clones of the third type – advanced version of the second type. For each clone type the method for detection is described. Proposed clone detection algorithm is based on program's abstract syntax tree (AST) analysis and, thus, it is accurate and does not have false positives. Existing clone detection tools are reviewed, their shortcomings are pointed out and it is shown that proposed method works correctly in considered situations. The approach described in this article has been implemented in the Klocwork inSight tool in refactorings "Extract method" and "Introduce variable". Finally, idea on improvement of the proposed approach are given: using of data flow analysis.

Keywords. refactoring, clone detection

References

[1]. M. Fowler., K. Beck, J. Brant, W. Opdyke, D. Roberts. Refactoring. Improve the design of exesting code. Addison-Wesley, 2001
[2]. http://www.semdesigns.com/products/clone/CCloneDR.html
[3]. http://www.ccfinder.net
[4]. T. Cormen, C. Leiserson, R. Rivest, K.Stein Introduction to Algorithms. MIT Press, 2002

Применение языка KAST для преобразования исходного кода и автоматического исправления дефектов

Н. Л. Луговской (lugovskoy@ispras.ru),
С. В. Сыромятников (syrom@ispras.ru)

Аннотация. В данной работе описывается расширение языка KAST для решения задачи трансформации исходного кода. В настоящее время язык KAST используется для поиска поддеревьев заданного в виде шаблона вида в синтаксических деревьях, построенных по коду на языках C/C++, Java и C#. В статье также рассматриваются некоторые существующие подходы к трансформации исходного кода и показываются преимущества использования для решения данной задачи языка KAST. Описывается метод, при помощи которого изменения в синтаксическом дереве преобразуются в изменения исходного кода.

Ключевые слова: трансформация кода; статический анализ; архитектура программ; язык шаблонов; KAST

1. Введение

Необходимость в трансформации исходного кода возникает во многих областях программной инженерии. Целями трансформации могут быть оптимизация, рефакторинг; она может применяться в обратной инженерии. Хотя преобразование кода обычно может быть проведено вручную, удобнее использовать для него специальные автоматические инструменты, работающие в соответствии с некоторой формальной спецификацией. Как правило, в качестве подобной спецификации используется набор правил модификации для структур данных компилятора (например, синтаксического дерева). В данной работе описывается подход, в котором спецификации трансформации задаются шаблонами на языке KAST.

KAST, язык написания шаблонов для синтаксических деревьев, был описан в статье [4]. Мы не будем подробно останавливаться на синтаксисе этого языка и принципах работы соответствующего анализатора, поскольку эта информация может быть найдена в [4]. Напомним лишь, что KAST предназначен для поиска поддеревьев заданного в виде шаблона вида в синтаксических деревьях, построенных по коду на языках C/C++, Java и C#, и

в настоящее время используется для выявления синтаксических структур, представляющих собой дефекты исходного кода. Однако синтаксис языка KAST после небольшого расширения позволяет использовать его для написания процедур трансформации кода; при этом шаблоны будут задавать фрагменты кода, подлежащие замене, а описываемые в данной статье расширения языка позволят определять те конструкции, на которые их следует заменять. Подобный механизм представляется весьма полезным для разработчиков, так как, с одной стороны, опирается на знание синтаксической структуры программы, а с другой — является легко настраиваемым в соответствии с потребностями конкретного пользователя.

В рассматриваемом подходе изменяемые фрагменты кода представляют собой узлы синтаксического дерева. Представляется логичным, чтобы преобразованный код также сначала конструировался из синтаксических узлов и лишь затем преобразовывался в текстовое представление. Поэтому требуемые расширения языка KAST сводятся к добавлению средств создания узлов синтаксического дерева (конструкторов) и функций, осуществляющих замену одного узла на другой, а также вставку и удаление узлов в синтаксическом дереве. Соответствующим образом доработанный KAST-анализатор в настоящее время существует в виде прототипа и, в отличие от обнаружителя дефектов, не входит в состав продуктов компании Klocwork. Однако полученные с его помощью результаты представляются авторам достаточно содержательными и достойными освещения в рамках отдельной статьи.

Проиллюстрируем возможности рассматриваемого подхода на нескольких примерах.

Пример 1. Заменить тип видимости для всех публичных полей данных в классе на приватный, одновременно добавив публичные методы для получения и установки значения данного поля.

Очевидно, что в результате преобразования из примера 1 мы вероятнее всего получим синтаксически некорректный код. Нам потребуются дополнительные изменения в коде, которые, вообще говоря, сложно свести к одному правилу. В частности, будет полезно такое автоматическое преобразование:

Пример 2. Заменить все непосредственные присваивания полям классов, осуществляемые не из методов этих классов, на функциональные вызовы. Имеются в виду вызовы функций установки значения полей, сгенерированные в результате преобразования из примера 1.

Пример 3. Преобразование кода в соответствии с гипотетическим Coding Style: добавить к именам всех параметров всех конструкторов префикс *i_*.

Преобразование соответствующего шаблону фрагмента кода можно рассматривать и в несколько ином аспекте. Если шаблон описывает некорректные языковые конструкции, то преобразование может состоять в наиболее вероятном исправлении обнаруженного дефекта. Разумеется,

предложенный вариант может не совпадать с тем, что на самом деле намерен написать пользователь, но рекомендация по возможному исправлению ошибки может оказаться полезной. Описываемый подход иллюстрирует

Пример 4. Найти в коде на языке C определения всех функций, для которых не указан тип возвращаемого значения, и добавить в эти определения ‘int’ в качестве типа результата. В языке C, в отличие от современных реализаций C++, отсутствие спецификации типа у функции или переменной означает, что функция возвращает значение типа ‘int’, а переменная, соответственно, принадлежит к этому типу. Подобные умолчания затрудняют понимание кода и делают его более трудным для сопровождения, а предлагаемое в данном примере автоматическое преобразование суть наиболее очевидное решение данной проблемы.

2. Существующие решения

Рассмотрим некоторые существующие инструменты трансформации исходного кода. Все описанные ниже системы работают по одному и тому же принципу: после синтаксического анализа кода к построенным структурам (синтаксическому дереву или его аналогам) применяется набор правил вида:

старая конструкция → новая конструкция

Далее по изменённому внутреннему представлению генерируется программный код.

Главное отличие предлагаемого нами подхода от данного принципа состоит в том, что в шаблоне на языке KAST можно определить не только конструкцию, подлежащую замене, но и контекст, в котором она должна находиться, чтобы быть заменённой: шаблон на языке KAST может описывать достаточно большое поддерево синтаксического дерева, и при этом лишь небольшая часть этого поддерева будет реально заменена. Кроме того, язык KAST позволяет обращаться к семантической информации внутри шаблона, что существенно повышает точность поиска требуемых синтаксических структур.

2.1. DMS: Software Reengineering Toolkit

DMS представляет из себя набор инструментов для автоматического анализа, трансформации и генерации программного кода ([1]). Система написана на языке PARLANSE, который позволяет производить символьные вычисления параллельно.

Преобразование исходного кода в DMS описывается набором правил. В общем виде правило преобразования может быть записано следующим образом:

```
LHS → RHS [ if_condition ]
```

, где *LHS* (левая сторона) и *RHS* (правая сторона) представляют собой шаблоны в виде завершенных конструкций языка программирования с метапеременными, а *if_condition* описывает условие, при котором должно осуществляться преобразование, в зависимости от значений переменных из *LHS*. Условие, в частности, может содержать вызовы функций, написанных на языке PARLANSE.

Типичное правило преобразования выглядит так:

```
default domain C.

rule auto_inc(v:lvalue):

statement->statement =

"\v = \v+1;" rewrites to "\v++;"

if no_side_effects(v).
```

В соответствии с данным правилом все конструкции вида *X=X+1* будут заменены на *X++*, где X - выражение, допустимое в качестве левой части оператора присваивания.

2.2. Stratego/XT

Stratego/XT ([2]) представляет собой программную среду для разработки систем трансформации кода. Она состоит из двух частей: Stratego, языка описывающего трансформацию программы, и XT - набора инструментов для трансформации.

Stratego/XT использует формат *Annotated Term* (*ATerm, аннотированный терм*) для представления синтаксических деревьев. Терм может быть числом, строкой, списком или *конструктором* вида *C(t1, ..., tn)*, где все *ti* суть тоже термы. Так, выражение *4 + f(5 * x)* представляется следующим термом формата ATerm: *Plus(Int("4"), Call("f", [Mul(Int("5"), Var("x"))]))*. Трансформация программ описывается с помощью правил, состоящих из терма-шаблона (определяющего, что надо заменить) и замещающего терма (соответствующего изменённому коду). Каждое правило имеет вид:

```
L: p1 -> p2
```

, где *L* - имя правила, *p1* - терм-шаблон, а *p2* - замещающий терм. Так, в соответствии с правилом

```
E : Eq(x, Int("0")) -> Not(x)
```

терм *Eq(Call("foo", []), False)* будет заменён на *Not(Call("foo", []))*, при этом переменной "x" будет сопоставлен терм *Call("foo", [])*.

3. Синтаксис языка KAST

Язык KAST позволяет описывать структуру поддеревьев синтаксического дерева в виде шаблонов. Шаблон, написанный на KAST'e, имеет следующий вид (##):

// Tun_1 **[** *Спецификация_1,1* **]** ... **[** *Спецификация_1,k1* **]** **/**

Квалификатор_1 **::** *Tun_2* **[** *Спецификация_2,1* **]** ... **[**

Спецификация_2,k2 **]**

...

Квалификатор_n **::** *Tun_n* **[** *Спецификация_n,1* **]** ... **[**

Спецификация_n,kn **]**

Здесь *курсивом* обозначены метапеременные, а **полужирным начертанием** выделены элементы языкового синтаксиса.

В (##) *Tun_i* - это имя одного из типов узлов дерева или звёздочка ('*'), что соответствует произвольному имени типа. *Квалификаторы* – это имена дочерних ссылок узла или некоторые предопределённые оси: *parent, ancestor, descendant, following-sibling*.

Спецификации могут быть присваиваниями (в них определяются переменные) и ограничениями (они суживают множество подходящих узлов). *Ограничения* могут содержать числовые значения и операции с ними, логические операции, строковые выражения и *подшаблоны*, которые соответствуют формату (##) за тем исключением, что начинаются сразу с квалификатора. В ограничениях могут использоваться вызовы функций, как предопределённых, так и определяемых пользователем. *Присваивания* служат для объявления и инициализации *переменных*; переменная может инициализироваться любым выражением, удовлетворяющим условиям на синтаксис ограничений.

Например, следующий шаблон соответствует всем присваиваниям, являющимся аргументами оператора sizeof (вида *sizeof (a = b)*):

// UnaryExpr [Op::Op [@Code = KTC_OPCODE_SIZEOF]]

/ Expr::ParensExpr / Expr::BinaryExpr [Op::Op [@Code =

KTC_OPCODE_ASSIGN]]

По сравнению с версией языка KAST, описанной в статье ([4]), в существующей его реализации добавилась поддержка *коллекций*. Коллекция представляет собой упорядоченную совокупность элементов различной природы. Каждый элемент коллекции является выражением произвольного

типа: численного, логического, семантического (ссылки на семантические элементы) или синтаксического (ссылки на узлы синтаксического дерева). Коллекция может быть как получена в результате вызова некоторой функции, так и задана явно при помощи фигурных скобок: *{1, 'abc'}*. Для работы с подобными совокупностями существует набор встроенных функций (добавления и удаления элементов, проверки наличия заданного элемента и т. п.), а также специальная конструкция for, позволяющая выполнить некоторую типовую операцию с каждым элементом. В нижеприведённом примере коллекция сначала явно определяется, а затем поэлементно распечатывается:

…

[$coll := { 1, 2, 'three', false() }]

[for($iter, $coll, $iter.println())]

Переменная *$iter* здесь является *итератором*, то есть каждый раз указывает на очередной элемент коллекции *$coll*.

В данной статье описываются расширения языка KAST, позволяющие создавать новые узлы, отсутствовавшие в изначально построенном синтаксическом дереве, и производить модификацию синтаксического дерева путём замены, добавления и удаления узлов.

4. Конструкторы узлов

Для создания новых узлов синтаксического дерева вводится оператор **new**. Его синтаксис подразумевает, что после ключевого слова ”new” идёт тип создаваемого узла, после которого в круглых скобках в произвольном порядке следуют инициализаторы дочерних ссылок и атрибутов:

new *TypeName***(***Child1* **:** *Node1***,**

…

ChildN **:** *NodeN***,**

@*Attribute1* **:** *Value1***,**

…

@*AttributeM* **:** *ValueM***)**

Нетрудно заметить, что синтаксис создания узла дерева в расширенном языке KAST схож с синтаксисом создания объекта в языках C++ и Java.

Проиллюстрируем сказанное на примере:

```
// IdExpr [ new BinaryExpr(Left : new LiteralExpr(@Value : 1),
                           @Op : KTC_OPCODE_ADD,
                           Right : this())
       ]
```

В результате применения данного шаблона для всякого выражения-идентификатора *X* будет построено бинарное выражение *1 + X* (константа *KTC_OPCODE_ADD* в языке KAST соответствует бинарному плюсу). Отметим, что фактического преобразования кода в соответствии с данным шаблоном не произойдёт, так как в этом шаблоне отсутствует вызов функции замены узлов в дереве.

Для некоторых дочерних ссылок и атрибутов возможны значения по умолчанию, которые используются, если данная ссылка или данный атрибут не были инициализированы явно. В случае ссылок значения по умолчанию допустимы для тех из них, где синтаксис языка программирования допускает отсутствие соответствующего данной ссылке поддерева (отсутствие аргументов у вызываемой функции, отсутствие инициализатора в декларации переменной и т. п.). К примеру, оператор new для создания (в терминах синтаксического дерева) вызова функции «foo» без аргументов может выглядеть так:

```
new CallExpr(Func : new IdExpr(@Value : 'foo'))
```

В случае, когда в конструируемом поддереве некоторое множество узлов должно быть объединено в однородный список, следует использовать коллекции. Конструктор вызова функции “foo” с параметрами “1” и “2” будет выглядеть так:

```
new CallExpr( Func : new IdExpr(@Value : ‘foo’),
              Args : {new LiteralExpr(@Value : 1), new
LiteralExpr(@Value : 2)}
            )
```

5. Функции модификации синтаксического дерева

Рассмотрим произвольное корневое ориентированное дерево (например, синтаксическое дерево, построенное по одной единице компиляции). Выделим в нём некоторую вершину N. Существует конечное непустое множество поддеревьев исходного дерева, для которых вершина N является корнем.

Среди этого множества поддеревьев существует максимальное. Очевидно, оно включает в себя все вершины, достижимые из вершины N в исходном ориентированном дереве. Описанное максимальное поддерево мы будем называть *листовым поддеревом (с корнем N).*

Описанные ниже функции модификации синтаксического дерева оперируют именно листовыми поддеревьями, а не отдельно взятыми узлами. Иными словами, при удалении или добавлении узла удаляется (соответственно, добавляется) не только сам узел, но и все его дочерние узлы, дочерние узлы его дочерних узлов и так далее. При замене одно листовое поддерево целиком подменяется другим.

Функция, осуществляющая замену одного (существующего) листового поддерева в синтаксическом дереве на другое (как правило, вновь созданное), имеет следующий синтаксис:

substitute(*Old_node***,** *New_node***)**

Дополним пример из предыдущей части:

```
// IdExpr [ $new_node := new BinaryExpr(Left : new
LiteralExpr(@Value : 1),
                                        @Op :
KTC_OPCODE_ADD,
                                        Right : this())
    ]
    [ substitute(this(), $new_node) ]
```

В соответствии с данным шаблоном всякое выражение-идентификатор *X* будет заменено на бинарное выражение *1 + X*.

В случае, когда некоторое листовое поддерево требуется заменить на несколько других поддеревьев, образующих в синтаксическом дереве однородный список, следует использовать коллекции:

substitute(*Old_node***, {***New_node_1***, …, ***New_node_n***})**

Функция вставки листового поддерева в дерево позволяет добавить поддерево в уже имеющийся в дереве однородный список узлов (например, список деклараций и операторов в теле функции). В качестве параметра указывается узел (т. е. корень некоторого другого поддерева), перед которым следует произвести вставку:

insert(*Node_after***,** *New_node***)**

Возможно добавление в список сразу нескольких поддеревьев, для чего также следует использовать коллекции:

insert(*Node_after***, {***New_node_1***, …,** *New_node_n***})**

Подчеркнём, что в описании всех четырёх приведённых выше функций *Old_node* и *Node_after* - это корневые узлы уже существующих поддеревьев, найденных в процессе сопоставления шаблона с конкретным синтаксическим деревом, а все *New_node* могут быть как вновь созданными поддеревьями, так и уже существующими, если требуется изменить их положение в дереве.

Удаление поддеревьев из дерева осуществляется при помощи функции **delete**:

delete(*Existing_node***)**

6. Модификация исходного кода

В результате работы функций модификации синтаксического дерева в него вносятся изменения, которые далее необходимо трансформировать в изменения программного кода. Важно не просто получить код для новых узлов дерева, но и правильно вставить его в исходную программу, сохранив пользовательский стиль, комментарии и макровызовы. Стоит заметить, что вновь созданные поддеревья конструируются из узлов двух видов: часть узлов берётся из исходного дерева, а часть создаётся с помощью конструкторов. При этом узлы, изначально существовавшие в исходном дереве, обязательно целиком образуют одно или несколько *листовых поддеревьев*, являющихся частью подобного нового поддерева.

Мы не будем вдаваться в подробности построения синтаксического дерева программы, они описаны в статье [3]; отметим лишь, что наряду с синтаксическим деревом в нашем инструменте сохраняется соответствующая этому дереву последовательность лексем исходного кода, которая обычно удаляется после стадии синтаксического анализа. При этом каждый узел синтаксического дерева имеет ссылку на свою начальную и конечную лексемы. Данный подход позволяет предельно точно сохранить изначальный стиль исходного кода, а также комментарии и макровызовы для каждого из узлов синтаксического дерева.

Рассмотрим, каким образом замена одного поддерева другим преобразуется в модификацию исходного кода программы, отметив, что изменение кода при вставке и удалении поддеревьев происходит аналогично.

Чтобы сохранить информацию о том, что в определенном месте синтаксического дерева некоторое листовое поддерево было заменено на новое, ребро, ведущее к этому поддереву, помечается специальной меткой. В метке указываются начальная и конечная лексемы старого поддерева. На рис.

1 показан пример, когда в выражении “*a = b + c*” происходит замена “*b + c*” на вызов функции “*bar()*”.

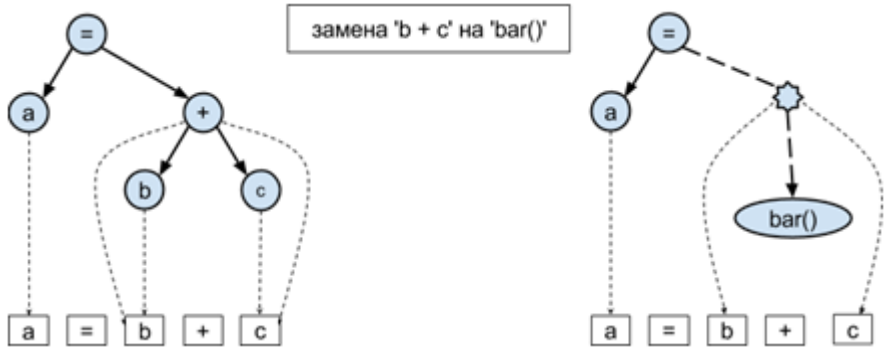


Рис. 1.

Ребро ведущее к узлу b + c помечается меткой L, а сам узел заменяется новым узлом, соответствующим вызову функции bar(). При этом последовательность лексем остается прежней, а в метку L лишь добавляется ссылки на первую и последнюю лексемы старого узла (“b + c”).

После того, как в дерево были внесены все изменения, достаточно совершить его обход и найти все метки. Очевидно, что если входящее в некоторое поддерево ребро уже помечено, то в этом поддереве не может быть других меток. Это значит, что изменения, описанные метками, не пересекаются, то есть их можно преобразовывать в изменения исходного кода в произвольном порядке (пересечений в исходном коде также не будет). Поэтому для каждой из меток:

- находится место вставки в исходный код (достаточно взять позиции из сохраненных для метки лексем)
- генерируется код для измененных частей поддерева

При генерации кода поддерева используется различный подход в зависимости от способа создания узла. Для узлов, созданных с помощью конструкторов, используется техника обхода и распечатки заранее заготовленных текстовых шаблонов для каждого конкретного типа узла. Для существующих узлов синтаксического дерева такой подход неприменим, поскольку не позволяет отобразить существующие узлы в том виде, в котором они присутствуют в исходном коде. Для них распечатывается последовательность лексем, заданная ссылками на начальную и конечную лексемы. В этой последовательности полностью сохранены изначальный стиль и комментарии, и это позволяет воспроизводить код для узлов рассматриваемого типа в неизменном виде.

Отдельно стоит упомянуть про сохранение макровызовов. Для того, чтобы информация о них присутствовала в синтаксическом дереве, существуют дополнительные ссылки от лексем, представляющих раскрытый макрос, на текстовый вид макровызова. Это дает возможность определить, какие лексемы были частью раскрытого макроса. Рассмотрим пример:

#define ADD(x, y) x + y

a = ADD(b, c);

Для данного кода будет построено синтаксическое дерево, изображённое на рис. 2.

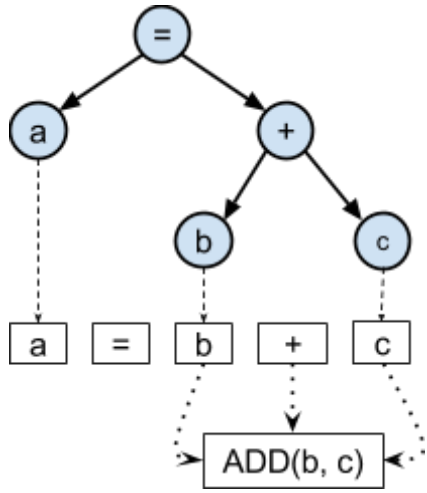


Рис. 2.

На рисунке также показаны лексемы, сопоставленные узлам дерева. Те из них, что находятся внутри раскрытого макровызова, имеют ссылку на его текстовое представление. Это позволяет сохранять изначальный вид макровызова при распечатке поддерева по последовательности соответствующих ему лексем.

7. Ограничения *метода и пути их преодоления*

1. Данный метод применим для преобразования кода исключительно в пределах одной единицы компиляции (одного исходного файла; в случае языков C/C++ - с подключёнными заголовочными файлами). Данное ограничение является достаточно принципиальным и не может быть преодолено в рамках описываемого подхода. Тем не

менее, класс задач, решаемых с применением предлагаемого метода, остаётся весьма широким.

2. Потенциальная неэффективность. Рассмотрим **пример 3** более внимательно. При наличии у конструктора нескольких параметров обход его тела будет осуществляться по-новой для каждого параметра. Данное ограничение теоретически преодолимо, но в силу ограничения 1 и исходя из общих соображений о применимости предлагаемого подхода, едва ли стоит считать его критическим.
3. Недостаточная надёжность. По сравнению со встроенными в программный продукт Klocwork видами рефакторинга ([3]), в рассматриваемом подходе в принципе не гарантируется, что полученный в результате преобразования код будет семантически эквивалентен первоначальному или хотя бы будет компилироваться без ошибок. Тем не менее, предлагаемый метод является очевидно более точным, чем стандартные средства текстовых редакторов, и легко настраиваемым под потребности конкретных пользователей.
4. Преобразование кода на основе преобразования синтаксического дерева имеет, помимо очевидных преимуществ, и свои недостатки. Так, существуют фрагменты кода, для которых синтаксическое дерево не строится из-за условных директив препроцессора, и трансформация этих фрагментов рассматриваемым в данной статье способом невозможна.

8. Заключение

Описанный нами язык позволяет в сравнительно простом и понятном для пользователя виде специфицировать весьма сложные трансформации синтаксического дерева и выгодно отличается от существующих инструментов аналогичного назначения. Так, использование шаблонов языка KAST позволяет определять подлежащие замене конструкции максимально точно, с учётом синтаксического и семантического контекста, в котором эти конструкции находятся. Кроме того, предлагаемые функции модификации синтаксического дерева позволяют конструировать новые поддеревья, произвольным образом включая в их состав узлы, уже имеющиеся в исходном дереве.

Использование расширенного языка KAST для автоматического исправления дефектов представляет большой интерес не только в качестве исследовательского эксперимента. Подобная возможность была бы востребована в реальных программных продуктах, осуществляющих статический анализ кода. Планируется включить поддержку описанных в данной статье языковых расширений в состав продуктов компании Klocwork.

Приложение

Реализуем на языке KAST примеры 2 и 4 из введения.

Пример 2'. Заменить все непосредственные присваивания полям классов, осуществляемые не из методов этих классов, на функциональные вызовы.

```
// BinaryExpr [ @Op = KTC_OPCODE_ASSIGN ]
        [ Left::MemberExpr [ isVariable() ]
                [ isClassMember() ]
                [ $me_code := @Op ]
                [ $var_name := getName() ]
                [ $object := Expr::* ]
        ]
        [ $assignment := Right::* ]
        [ $func_name := concat('set_', $var_name) ]
        [ $new_func := new MemberExpr(Expr : $object,
                                Name : new Name (
@Id : $func_name ),
                                @Op : $me_code) ]
        [ substitute(this(), new CallExpr(Func : $new_func,
Args : { $assignment })) ]
```

Пример 4'. Найти в коде на языке C определения всех функций, для которых не указан тип возвращаемого значения, и добавить в эти определения 'int' в качестве типа результата.

```
// FuncDef [ isCLanguage() ]
        [ not DeclSpecs[*]::AnyTypeOf ]
        [ not DeclSpecs[*]::ReservedTypeSpec ]
        [ not DeclSpecs[*]::ClassType ]
```

[not DeclSpecs[*]::EnumType]

[not DeclSpecs[*]::TypeName]

[not DeclSpecs[*]::AutoType]

[$declarator := Declarator::]*

[$body := FuncBody::]*

[$declspecs := add-element(getDeclarationSpecifiers(),

new BuiltinType(@Spec :

KTC_BUILTINTYPE_INT))]

[substitute(this(), new FuncDef(DeclSpecs : $declspecs,

Declarator : $declarator,

FuncBody : $body))]

Список литературы

[1]. http://www.semdesigns.com/Products/DMS/DMSToolkit.html

[2]. http://strategoxt.org/Tools/WebHome

[3]. Н. Л. Луговской. Подход для проведения рефакторинга «Выделение функции» в инструменте Klocwork Insight. Сборник трудов Института системного программирования РАН. Под ред. акад. РАН Иванникова В. П. Т. 23. М., ИСП РАН, 2012. 476 с., с. 107-132.

[4]. С. В. Сыромятников. Декларативный интерфейс поиска дефектов по синтаксическим деревьям: язык KAST. Сборник трудов Института системного программирования РАН. Под ред. акад. РАН Иванникова В. П. Т. 21. М., ИСП РАН, 2011. 296 с., с. 51-68.

Source code transformation and automatic correction of defects with KAST language

N. L. Lugovskoy, S. V. Syromyatnikov
lugovskoy@ispras.ru, syrom@ispras.ru
ISP RAS, Moscow, Russia

Abstract. This article is devoted to KAST pattern language extensions introduced for purposes of source code transformation. Currently KAST is used for matching syntactic patterns in syntactic trees built of C/C++, Java or C# sources. After introducing several new functions that can be called from a pattern, KAST patterns became applicable not only for detecting desired fragments of syntactic trees, but for specifying modifications of those fragments as well. Several existing approaches to code transformation are also considered. The main advantage of the described KAST based technique over those approaches is that our patterns can describe constructs we need to change together with the context these constructs are used in. Another important benefit is that KAST patterns can deal with semantic information about tree nodes. This significantly increases precision of pattern matching. The article also describes a method for converting modifications of syntactic trees into modifications of source code. The described technique allows to preserve comments, macro calls and indentation that were used in the initial code fragment. Though our approach is applicable only to a single compilation unit and doesn't guarantee syntactic correctness of newly generated code, it can greatly ease the task of automatic code transformation.

Keywords: refactoring; code transformation; static analysis; program architecture; pattern language; KAST

References

[1]. http://www.semdesigns.com/Products/DMS/DMSToolkit.html
[2]. http://strategoxt.org/Tools/WebHome
[3]. N. L. Lugovskoj. Podkhod dlya provedeniya refaktoringa «Vydelenie funktsii» v instrumente Klocwork Insight [“Extract Function” Refactoring in Klocwork Insight Toolkit]. Trudy ISP RAN [The Proceedings of ISP RAS]. 2012, vol. 23, pp. 107-132 (in Russian).
[4]. S. V. Syromyatnikov. Deklarativnyj interfejs poiska defektov po sintaksicheskim derev'yam: yazyk KAST [Declarative Interface of Detecting Defects on Syntax Trees: KAST Language]. Trudy ISP RAN [The Proceedings of ISP RAS]. 2011, vol. 20, pp. 51-68 (in Russian).

Подход к обнаружению ошибок несоответствия типов в коде на динамических языках программирования

Бронштейн И. Е.
ibronstein@ispras.ru

Аннотация. Статья посвящена обнаружению дефектов в коде, написанном на динамических языках программирования. Производится обзор статических анализаторов программ на языках Python, Ruby и JavaScript. Показывается, что большинство существующих средств не в состоянии обнаруживать целый класс дефектов: ошибки несоответствия типов. Даётся определение таких ошибок, приводятся данные о том, что они весьма распространены, а также критичны по мнению разработчиков программ. Предлагается подход к выводу типов для динамических языков, а также к реализации обнаружителей дефектов на его основе. Вводится понятие трассы дефекта, описывается построение такой трассы.

Ключевые слова: статический анализ; обнаружение дефектов; динамическая типизация данных; Python; Ruby; JavaScript; несоответствие типов; трассы дефектов.

1 Введение

Среди существующих языков программирования можно выделить большую группу *динамических языков*. Главный фактор, определяющий, является ли язык динамическим, — используемая в языке типизация данных [1]. Статическая типизация данных означает, что контроль типов осуществляется во время компиляции. Напротив, при динамической типизации проверки производятся в процессе выполнения программы. В этом случае тип имеется у тех или иных значений, однако у переменных он как таковой отсутствует. В процессе работы программы одной и той же переменной могут быть присвоены значения различных типов [2].

Несмотря на отсутствие единого определения динамических языков программирования, обычно к ним относят языки, которые поддерживают: (1) динамическую типизацию данных; (2) изменение кода программы во время выполнения (например, добавление полей класса, изменение иерархии типов и

т. д.), а также выполнение произвольного кода «на лету» (функция `eval`). Как правило, динамические языки являются интерпретируемыми.

В последнее время такие языки получили большое распространение. Согласно индексу TIOBE по состоянию на август 2013 года [3], в десятку самых популярных языков программирования входят PHP, Python, JavaScript и Ruby, являющиеся динамическими языками. Perl также не сильно уступает им в популярности. Всё чаще используются фреймворки, включающие в себя динамические языки: AJAX (включает JavaScript), LAMP (Perl, PHP или Python), Ruby on Rails (Ruby). В последние годы динамические языки стали поддерживаться основными производителями интегрированных средств разработки (IDE). Популярность динамических языков можно проследить на примере языка Python: в последнее время он стал использоваться не только для написания небольших скриптов, но и для создания крупных программных продуктов. Исходный код таких проектов, как TACTIC, Twisted, Django и SQLAlchemy, содержит более сотни тысяч строк, что весьма велико, учитывая, насколько высокоуровневыми являются программные конструкции языка Python.

2 Обзор статических анализаторов для динамических языков

В связи со всё возрастающей сложностью программ, разрабатываемых на динамических языках, остро стоит проблема их надёжности. Чтобы решить эту проблему для программ на статических языках (например, Си/Си++ и Java), активно используются средства, позволяющие проводить анализ программ с целью обнаружения тех или иных дефектов. Для выявления ошибок реализации чаще всего используются методы статического анализа, осуществляемого по исходному коду анализируемой программы без её запуска. Исследования на конкретных программных продуктах показывают, что сложность исправления ошибки, обнаруженной при помощи статического анализа, ниже, чем если бы эта ошибка была найдена на более поздних этапах разработки [4].

Так, для программ на языках Си и Си++ существует множество профессиональных статических анализаторов (Svace [5], Coverity Prevent, Klocwork Insight, PVS-Studio и другие). Эти средства включают в себя набор автоматических обнаружителей дефектов (чекеров), которые могут: (1) выполняться после стадии семантического анализа путём обхода синтаксического дерева (AST) и анализа навешанных на это дерево атрибутов (без анализа путей выполнения); (2) осуществлять анализ потоков данных (dataflow) по промежуточному представлению программы. Эти две основные группы можно назвать простыми и, соответственно, сложными чекерами. С помощью сложных чекеров возможно обнаруживать критические дефекты: выход за границы массива, разыменование нулевого указателя, использование освобождённой памяти или значения неинициализированной переменной.

Для программ на динамических языках программирования также, хотя и в меньшем количестве, существуют статические анализаторы. Подавляющее большинство среди них составляют программные средства, не использующие вывод типов: например, Pyflakes [6] и Pylint [7] (для языка Python), reek [8] и ruby-lint [9] (для языка Ruby), JSLint [10] и JSHint [11] (для языка JavaScript). Все эти инструменты объединяет то, что в них реализован набор чекеров, выполняющихся путём обхода синтаксического дерева без навешанных на него атрибутов. В качестве примера можно привести нахождение операторов без побочного эффекта. Ниже приведены код на языке Python и предупреждение, которое выводится при анализе данного кода анализатором Pylint:

```
x = 0
x
```

```
$ pylint true_positive1.py
W:  2, 0: Statement seems to have no effect
(pointless-statement)
```

В целом, реализованные в вышеперечисленных средствах чекеры направлены на поиск в анализируемом коде некоторых анти-паттернов программирования. Иногда также проверяется, соответствует ли программа принятым в языке стандартам кодирования (например, PEP8 [12] для языка Python). Проводя аналогию с Си++, описанные чекеры можно назвать простыми. Они чаще всего соответствуют не сообщениям об ошибках, а предупреждениям, который выдавал бы компилятор статического языка программирования.

Хотя про выше названные инструменты сказано, что в них не реализован вывод типов, это не совсем так. В Pylint частично реализован вывод типов, на основе которого осуществляется проверка, можно ли использовать объект в выражении «вызов функции». Однако, вывод типов в Pylint является локальным (производится лишь в рамках одной функции) и неполным (поддерживает не все возможные выражения). Так, для следующего примера инструмент выдаст сообщение об ошибке:

```
x = 0
x()
```

```
$ pylint true_positive2.py
E:  2, 0: x is not callable (not-callable)
```

Однако, если пример немного усложнить (назовём получившийся пример `false_negative.py`), ошибка не будет обнаружена, поскольку вывод типов в Pylint не сопоставляет параметры функции и аргументы её вызова:

```
def foo(p): p()
x = 0
foo(x)


$ pylint false_negative.py
$
```

3 Ошибки несоответствия типов

Как видно, код в `false_negative.py` содержит ошибку, которая не обнаруживается ни одним из средств, представленных выше. Обобщая ситуацию, можно заметить, что поскольку в этих средствах не реализован вывод типов, они не могут обнаруживать *ошибки несоответствия типов*. Говоря неформально, под такими ошибками мы будем понимать непредусмотренный разработчиком вызов функции или операции (например, доступа к атрибуту объекта) с аргументами (операндами) типов, которые функцией (операцией) не поддерживаются. Ошибка в `false_negative.py` подходит под это определение, поскольку выражение `x()` можно считать неявным обращением к `x.__call__` (аналог метода `operator()` в Си++). К ошибкам несоответствия типов можно отнести и ситуацию, когда число параметров функции и число аргументов её вызова не совпадают.

Возникает вопрос, насколько важными являются именно ошибки несоответствия типов, и, следовательно, насколько оправдано решать задачу вывода типов, необходимого для их обнаружения. Чтобы ответить на этот вопрос, сначала определим исследуемый класс ошибок более строго.

Номер	Тип дефектов	Описание	Исправление
010	Документация	Комментарии, сообщения	Изменение документации, комментариев и т. д.
020	Синтаксис	Синтаксические ошибки, опечатки в коде и т. д.	Исправление синтаксической или иной ошибки во время компиляции
030	Сборка, пакеты	Подключение библиотек, контроль версий и т. д.	Создание или использование правильной версии
040	Присваивания	Декларации, дубликаты имён, области видимости имён и т. д.	Исправление оператора
050	**Интерфейс**	**Вызовы процедур, ввод/вывод, пользовательские форматы**	**Изменение интерфейса**
060	Проверки	Сообщения об ошибках, некорректные проверки	Добавление или исправление обработки ошибок
070	Данные	Структура, содержимое	Изменение программы для поддержания целостности данных
080	Функция	Логика, указатели, циклы, рекурсия, вычисления и т. д.	Изменение совокупности операторов
090	Система	Конфигурация, хронометраж, память	Адаптация программы к внешним факторам (например, ОС)
100	Окружение	Ошибки в системе поддержки: компиляторе, тестовых данных и т. д.	Исправление ошибки компилятора или отказ от его использования, исправление тестовых данных

Таблица 1. Общая DTS-классификация.

Рассмотрим стандарт Personal Software Process Defect Type Standard (PSP DTS) — распространённую классификацию программных дефектов [13,14]. В ней описаны десять различных видов ошибок, которые обычно встречаются в программных продуктах. Для каждого вида перечислены его номер (от 010 до 100), название (тип дефектов), описание, а также исправление, которое необходимо внести для устранения ошибок данного вида. Общая (независимая от языка программирования) DTS-классификация представлена в таблице 1.

Номер	Тип дефектов	Описание
010	Документация	Ошибки в документационных строках и комментариях
020	Синтаксис	SyntaxError, IndentationError
030	Стандарт кодирования	Предупреждения и ошибки, связанные с PEP8
040	Присваивание	NameError (нет объявления), IndexError/KeyError,
		UnboundLocalError (область видимости), ImportError
050	**Интерфейс**	**TypeError, AttributeError: неправильные параметры и методы**
060	Проверки	AssertionError (ошибка в assert), ошибки в тестах (doctest или unittest)
070	Данные	ValueError (неправильные данные), ArithmeticError,
		EOFError, BufferError
080	Функция	RuntimeError и логические ошибки
090	Система	SystemError (MemoryError, ReferenceError)
100	Окружение	EnvironmentError: ошибки ОС и сторонних библиотек, ошибки ввода/вывода (IOError)

Таблица 2. DTS-классификация для языка Python.

Как следует из общей DTS-классификации и определения ошибок несоответствия типов, последние соответствуют DTS-ошибкам вида 050 («интерфейс»). В терминах DTS несоответствие типов — это неправильное использование интерфейса, а исправление такой ошибки — это изменение либо интерфейсной функции, либо способа её использования (передаваемых в функцию аргументов).

Также существуют модификации DTS применительно к различным языкам программирования. В этой связи интерес представляет DTS-классификация для языка Python [15], представленная в таблице 2. В качестве описания для большинства видов дефектов приведены типы (классы) исключений, соответствующие данному виду дефектов. Ошибкам вида «интерфейс»

соответствуют исключения `AttributeError` и `TypeError`. Таким образом, с формальной точки зрения, ошибка несоответствия типов в языке Python — это ситуация, когда в результате вызова некоторой функции или метода возбуждается исключение типа `AttributeError` или `TypeError`, причём обработки этого исключения выше по стеку не происходит (это и означает, что вызов не был предусмотрен разработчиком), что приводит к аварийному завершению программы.

Для других динамических языков можно дать аналогичные определения с той лишь разницей, что типы исключений здесь будут другими. Так, для Ruby речь будет идти об исключениях классов `NoMethodError` и `TypeError`, для JavaScript — класса `TypeError`.

Наличие формального определения для ошибок несоответствия типов позволяет определить, насколько они важны, с помощью количественных критериев. Будем рассматривать два следующих критерия:

- долю отчётов об ошибках несоответствия типов среди всех отчётов об ошибках (распространённость);
- долю критичных отчётов среди отчётов об ошибках несоответствия типов по сравнению с долей критичных отчётов целом (критичность).

В качестве источника отчётов выберем системы отслеживания ошибок известных проектов с открытым исходным кодом на языке Python: Exaile [16], calibre [17], SQLAlchemy [18], Bazaar [19], Twisted [20], Trac [21] и Django [22].

Название проекта	**Все отчёты об ошибках**	**AttributeError и TypeError**	**Доля, %**
Exaile	1350	86	**6,37**
calibre	1357	77	**5,67**
SQLAlchemy	2826	230	**8,14**
Bazaar	4646	167	**3,46**
Twisted	6664	197	**2,96**
Trac	10472	627	**5,99**
Django	20664	969	**4,69**

Таблица 3. Распространённость ошибок несоответствия типов.

Данные по распространённости ошибок несоответствия типов приведены в таблице 3. Из таблицы видно, что доля отчётов, посвящённых именно ошибкам несоответствия типов, для некоторых проектов превышает 8 %. При этом необходимо учитывать, что тема значительной части отчётов (более 90 %) —

не дефекты, а логические ошибки, которые обнаружить автоматическими средствами невозможно [2]. Таким образом, отчёты об ошибках несоответствия типов составляют для некоторых проектов большинство отчётов об ошибках, которые можно найти автоматически.

Название проекта	Критичность всех отчётов, %	Критичность AE/TE, %	Более критичны
Exaile	8,88	11,63	**Д**
Bazaar	4,72	4,25	Н
Trac	5,36	10,03	**Д**
Django	1,95	2,68	**Д**

Таблица 4. Критичность ошибок несоответствия типов.

Данные по критичности ошибок несоответствия типов приведены в таблице 4. Критичными считались отчёты с большим значением поля «критичность», если таковое присутствовало в системе отслеживания ошибок: «Critical» и «High» в поле «Importance» (для Exaile), «Critical» в поле «Importance» (для Bazaar), «blocker» и «critical» в поле «Severity» (для Trac), «Release blocker» в поле «Severity» (для Django). Для всех проектов, кроме Bazaar, критичность ошибок несоответствия типов выше, чем в среднем. Заметим, однако, что 15 % кода Bazaar написано на языке Си. Это могло послужить причиной более низкой критичности ошибок несоответствия типов для этого проекта.

4 Обнаружение ошибок в программах на динамических языках

Таким образом, ошибки несоответствия типов широко распространены и, по мнению самих разработчиков проектов, достаточно критичны. Для обнаружения таких ошибок необходимо наличие глобального и полного (в смысле поддержки программных конструкций того или иного динамического языка) вывода типов. Отметим, что, несмотря на отсутствие такого вывода типов в перечисленных выше статических анализаторах, исследовательские проекты по реализации вывода типов для различных динамических языков всё же существуют. К подобным проектам можно отнести DRuby [23] (для языка Ruby) и TAJS [24] (для языка JavaScript).

В данной работе представляется подход к построению статического анализатора, обнаруживающего ошибки несоответствия типов на основе информации, полученной на этапе вывода типов. Для вывода типов предлагается использовать алгоритмы на графе типовых переменных. Работа с таким графом во многом аналогична другим методам вывода типов, например, используемым в проекте DRuby уравнениям, задающим ограничения на типы

выражений [25]. Однако, на наш взгляд, представление ограничений именно в виде графа значительно упрощает построение трасс дефектов, о которых будет сказано несколько позже.

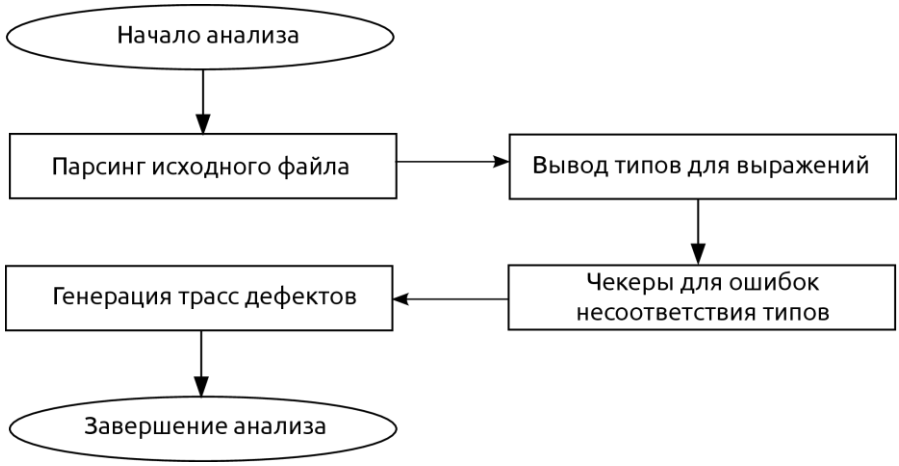


Рис. 1. Общая схема работы статического анализатора.

Общая схема работы статического анализатора в рамках предлагаемого подхода представлена на рис. 1. Ниже мы более подробно разберём все этапы работы за исключением парсинга исходного кода. Задача парсинга, как правило, не представляет особой сложности и может быть решена как стандартными средствами языка (например, модуль `ast` для языка Python), так и при помощи сторонних библиотек (например, jruby-parser [26] для языка Ruby или Esprima [27] для языка JavaScript).

5 Вывод типов для выражений

В предлагаемом подходе к построению статического анализатора для вывода типов используется представление программы в виде графа типовых переменных. Каждому выражению в программе ставится в соответствие свой узел графа: типовая переменная, то есть переменная, значением которой является множество типов. Изначально типовые переменные хранят в себе пустые множества типов, за исключением констант и строковых литералов, которые в качестве значения получают множество из одного единственного типа: типа константы (литерала). Множество возможных типов для выражения `expr` обозначается `type(expr)`.

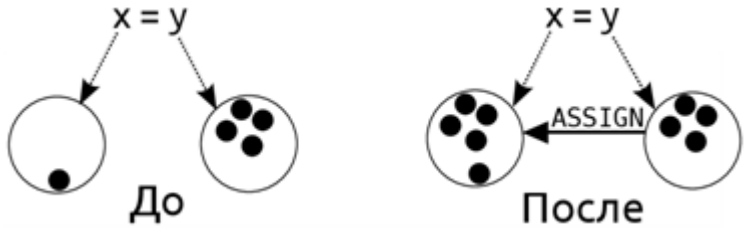


Рис. 2. Ситуация до и после добавления дуги и пропагации по ней типов.

Граф типовых переменных является ориентированным. Дуги графа, называемые также ограничениями на типы, создаются при обходе тех или иных языковых конструкций в программном коде. Дуги помечаются названием: видом ограничения, который зависит от того, какая конструкция анализируется (например, для присваивания соответствующее ограничение будет иметь вид `ASSIGN`). Типы распространяются (пропагируются) по графу типовых переменных по определённым правилам, которые устанавливаются ограничениями. Так, если в программе выполняется присваивание `x = y`, то любой тип, который может принимать выражение `y`, может также иметь и переменная `x`. Поэтому ограничение вида `ASSIGN` пропагирует все типы из `y` в `x`, отражая тот факт, что $\texttt{type(y)} \subseteq \texttt{type(x)}$. Это иллюстрирует рис. 2 (типы изображены на нём в виде чёрных точек).

Добавление в граф новых ограничений создаёт необходимость в дальнейшей пропагации типов. Верно и обратное: например, для вызова некоторого метода в результате пропагации может быть вычислен новый тип объекта. Значит, возможно, будет необходимо анализировать тела методов в этом объекте, а при анализе тел методов могут быть созданы новые дуги в графе (ограничения) и так далее. В алгоритмах, которые работают с графом типовых переменных, циклически создаются новые ограничения и пропагируются типы до тех пор, пока эти действия вносят изменения в граф. Работа с графом типовых переменных зависят от семантики того или иного динамического языка программирования. Виды ограничений, создаваемых для программ на языке Python, подробно описаны в статье [2], которая служит основой для исследовательского проекта TIRPAN [28].

В выше упомянутой работе также предлагается подход к анализу функций, написанных не на языке Python и таким образом являющихся внешними по отношению к анализируемой программе. Суть подхода, который можно распространить и на другие динамические языки программирования, в следующем: поведение внешних функций моделируется в статическом анализаторе при помощи *спецификаций*. Спецификации — это специальные функции, которые принимают на вход типы параметров исходной функции, а

на выходе по ним строится тот тип, который имело бы возвращаемое значение исходной функции. Также, возможно, моделируется внесение изменений в типы входных параметров (например, для функции `append` в языке Python или аналогичной ей функции `push` в языке Ruby тип первого параметра меняется в ходе выполнения).

6 Чекеры для ошибок несоответствия типов

Стадия работы чекеров следует непосредственно после вывода типов. Входными данными для чекеров является AST с навешанными на него типовыми переменными, хранящими информацию о типах выражений. Однако, в данной работе предполагается, что чекеры работают с AST не напрямую, а посредством специально предназначенной для этого прослойки: фреймворка для чекеров. Ниже мы опишем функциональность этого фреймворка.

Фреймворк осуществляет итерирование по AST, при этом пропускаются участки кода, для которых выше по стеку есть обработчики исключений, соответствующих ошибкам несоответствия типов для данного языка. Например, для следующего кода на языке Ruby фреймворк пропустит код с вызовом `getTranslation`, иначе это могло бы привести к многочисленным ложным срабатываниям чекера, проверяющего доступ к несуществующему методу:

```
begin
    translation = message.getTranslation()
rescue NoMethodError
    translation = message
end
```

Фреймворк предоставляет чекерам возможность зарегистрировать произвольное число функций обратного вызова (callback), которые будет вызываться при входе в узлы или выходе из узлов определённого типа. Предположим, что для кода на языке Python необходимо реализовать чекер `basenameChecker`, проверяющий, что в функцию `os.path.basename` было передано значение, отличное от строки. Регистрация такого чекера может выглядеть следующим образом:

```
def basenameChecker():
  registerCallback(Event.ON_ENTER, ast.Call,
processBasenameCall)
```

Основная функциональность фреймворка — это предоставление интерфейсных функций следующих видов: (1) функций, возвращающих множество типов для узла (например, `getNodeTypes`); (2) предикатов для проверки типа (например, `isFunction`); (3) функций для работы с внутренней структурой типа или для вывода его внешнего представления (например, `getListElements` или `getQualifiedName`); (4) функций формирования сообщения об ошибке (например, `reportError`). Такой интерфейс позволяет реализовать каждую из функций, из которых состоит чекер несоответствия типов, в компактном виде. Например, реализация `processBasenameCall` с использованием интерфейсных функций может выглядеть так:

```
def isBasenameFunction(element):
    return (isFunction(element) and
            getQualifiedName(element) ==
"os.path.basename")


def processBasenameCall(node):
    functionTypes = node.func.getNodeTypes()
    if any(isBasenameFunction(type) for type in
functionTypes):
        argTypes = node.args[0].getNodeTypes()
        for type in argTypes:
            if not isString(type):
                reportError(Defect.BASENAME,
node, type)
    return TRAVERSE_ALL
```

7 Генерация трасс дефектов

Предлагаемый подход к построению статического анализатора предполагает генерацию трасс для найденных ошибок несоответствия типов. Говоря неформально, трасса дефекта — это описание последовательности шагов, из которой можно понять, как неправильный тип попадает в точку несоответствия типов. Такое описание должно помочь пользователю статического анализатора определить, является ли обнаруженный дефект истинным. Чтобы понять важность наличия трассы, рассмотрим ошибку несоответствия типов из проекта Gramps [2, 29], которая заключается в том, что в функцию `os.path.basename` вместо строки передаётся пустой список:

```
class HtmlDoc:

...
    def set_css_filename(self, css_filename):
        if os.path.basename(css_filename):
            self.css_filename = css_filename
        else:
            self.css_filename = ''


class DocReportDialog:
    def __init__(self):
        self.doc = HtmlDoc(...)
        self.CSS =
PLUGMAN.process_plugin_data("WEBSTUFF")
        self.css_filename =
self.CSS[id]["filename"]
...
    def make_document(self):


self.doc.set_css_filename(self.css_filename)
```

```
...
def on_ok_clicked(...):
    dialog = DocReportDialog(...)
    dialog.make_document()
```

При обнаружении в `set_css_filename` дефекта несоответствия типов чекер `basenameChecker` может предоставить пользователю трассировку стека (stack trace), включающую в себя функции `set_css_filename`, `make_document`, `on_ok_clicked` и так далее. Однако, такая трассировка малоинформативна, поскольку не позволяет проследить, откуда у `css_filename` появилось значение, не являющееся строкой. Эту информацию можно получить путём «обратного разбора» функции `process_plugin_data`, что, с свою очередь, потребует дальнейшего анализа. Подобный анализ и является генерацией трассы дефекта.

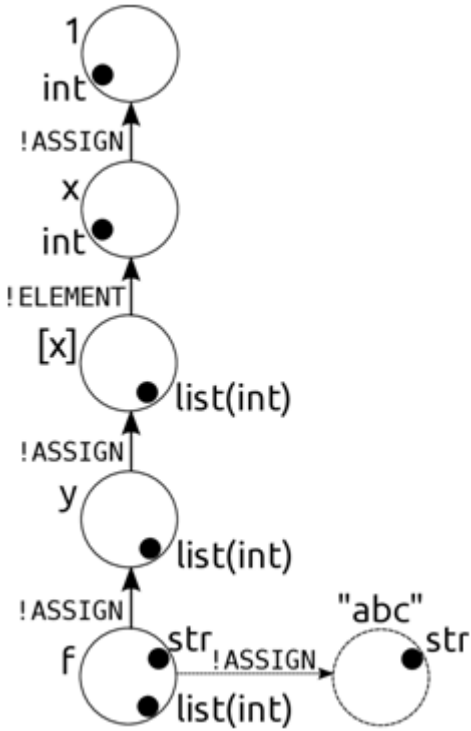


Рис. 3. «Обратный» граф типовых переменных.

Так как типовые переменные являются вершинами ориентированного графа, понятие трассы дефекта можно определить более формально, рассмотрев новый граф, в котором все вершины соединены обратными дугами. Это означает, что, если в исходном графе имелось ограничение вида `ASSIGN` от переменной `A` к переменной `B`, в новом графе будет присутствовать дуга от `B` к `A`. Будем считать, что обратные дуги имеют вид «конкатенация `!` и названия вида первоначальной дуги», например, `!ASSIGN`.

После того, как мы ввели новый граф, трассу можно определить как набор путей в нём, причём каждый путь начинается в переменной, соответствующей месту дефекта (например, аргументу `os.path.basename` для `Defect.BASENAME`) и строится по определённым правилам, которые задаются дугами в графе. Рассмотрим небольшой участок кода и покажем на примере, как при помощи его «обратного» графа (показан на рис. 3) для дефекта `Defect.BASENAME` строится трасса, состоящая из ровно одного пути:

```
x = 1
y = [x]
if condition(): f = y
else: f = "abc"
print os.path.basename(f)
```

Построение пути начинается с типовой переменной для `f`, которая становится текущей вершиной для алгоритма. Для каждого узла в алгоритме задаётся текущее множество типов, поиск которых осуществляется. Первоначально (для первой вершины пути) это множество равно множеству «неправильных» типов для дефекта. Для данного примера множество равно `{list(int)}`. Далее производится добавление в трассу новых вершин графа в соответствии с ограничениями и обход вершин в глубину. Ограничение вида `!ASSIGN` добавляет в граф вершину на конце дуги (назовём её $V_{!ASSIGN}$) в случае, если пересечение множества искомых типов для текущего узла и $type(V_{!ASSIGN})$ не пусто. Когда $V_{!ASSIGN}$ станет текущей вершиной, множеством искомых типов для неё будет это пересечение. Согласно этому правилу, переменная для `"abc"` в трассу не добавляется (поэтому она и дуга к ней изображены на рис. 3 пунктиром), а переменные для `y` и `[x]` добавляются. Правило для вершин на конце `!ELEMENT` также вычисляет пересечение типов, но при этом «раскрываются» коллекции (например, `{list(int)}` превращается в `{int}`). По этому правилу в трассу добавляется переменная для `x` и, следовательно, для `1`. На этом построение трассы заканчивается, поскольку вершин для добавления в неё больше нет.

8 Заключение

В работе предложен подход к построению статического анализатора, обнаруживающего в коде на динамических языках программирования ошибки несоответствия типов и генерирующего трассы для найденных дефектов. При этом описанные методы не привязаны к специфике какого-либо конкретного динамического языка.

Перечислим возможные направления дальнейших исследований по рассматриваемой тематике:

1. Исследование того, насколько чувствительность к потоку выполнения важна для повышения точности анализа (в настоящий момент алгоритмы, работающие с графом типовых переменных, нечувствительны к потоку выполнения).
2. Добавление поддержки инкрементального анализа, при котором небольшие изменения в тексте программы вызывают пересчёт небольшого числа типов выражений, что позволило бы выполнять статический анализ при наборе кода в IDE.
3. Организация обратной связи, которая позволила бы использовать при анализе не только вычисленную информацию, но и некоторые подсказки от пользователя о возможных типах выражений в программе.

Список литературы

[1]. L. D. Paulson. Developers shift to dynamic programming languages. IEEE Computer, vol. 40, issue 2, February 2007, pp. 12–15.

[2]. Бронштейн И. Е. Вывод типов для языка Python. Труды Института системного программирования РАН, том 24, 2013, стр. 161–190.

[3]. TIOBE Programming Community Index for August 2013: http://tinyurl.com/tiobe-201308

[4]. Савицкий В. О., Сидоров Д. В. Инкрементальный анализ исходного кода на языках C/C++. Труды Института системного программирования РАН, том 22, 2012, стр. 119—129.

[5]. Аветисян Арутюн, Белеванцев Андрей, Бородин Алексей, Несов Владимир. Использование статического анализа для поиска уязвимостей и критических ошибок в исходном коде программ. Труды Института системного программирования РАН, том 21, 2011, стр. 23–38.

[6]. Pyflakes: https://launchpad.net/pyflakes

[7]. Pylint - code analysis for Python: http://www.pylint.org

[8]. reek: https://github.com/troessner/reek

[9]. ruby-lint: http://code.yorickpeterse.com/ruby-lint/latest

[10]. JSLint: http://www.jslint.com

[11]. JSHint, a JavaScript Code Quality Tool: http://www.jshint.com

[12]. PEP 8 – Style Guide for Python Code: http://www.python.org/dev/peps/pep-0008

[13]. R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Y. Wong. Orthogonal Defect Classification — A Concept of In-Process Measurements (1992).

[14]. W. Humprey. A discipline for software engineering (1997).

[15]. M. Reingart. Rad2py: Plataforma de Trabajo para el Desarrollo Rápido de Aplicaciones bajo un Proceso de Software Personal, pp. 152–153: http://code.google.com/p/rad2py/wiki/DefectTypeStandard
[16]. Bugs : Exaile: https://bugs.launchpad.net/exaile
[17]. Bugs : calibre: https://bugs.launchpad.net/calibre
[18]. sqlalchemy: http://www.sqlalchemy.org/trac
[19]. Bugs : Bazaar: https://bugs.launchpad.net/bzr
[20]. Twisted: http://twistedmatrix.com/trac
[21]. The Trac Project: http://trac.edgewall.org
[22]. Django: https://code.djangoproject.com
[23]. DRuby - Home: http://www.cs.umd.edu/projects/PL/druby
[24]. Type Analysis for JavaScript: http://www.brics.dk/TAJS
[25]. Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, Michael Hicks. Static type inference for Ruby. SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing, pp. 1859–1866.
[26]. jruby-parser: https://github.com/jruby/jruby-parser
[27]. esprima: https://github.com/ariya/esprima
[28]. tirpan: https://github.com/bronikkk/tirpan
[29]. Gramps Bug Report 005023: http://www.gramps-project.org/bugs/view.php?id=5023

Approach to detecting types inconsistency errors in a program code in dynamic languages

I. E. Bronshteyn
ibronstein@ispras.ru

Abstract. The paper deals with detection of defects in a program code written in dynamic languages. At first, overview of static analyzers for programs in Python, Ruby and JavaScript is done. After this, the paper shows that most of existing tools are not able to detect entire class of defects: types inconsistency errors. Such errors are defined, the paper proves that the errors are prevailing and rather critical in the opinion of software developers. It presents an approach to type inference for dynamic languages and to implementation of checkers based on output from type inference. Concept of defect trace is introduced and construction of such trace is described then.

Keywords: static analysis; defects detection; dynamic typing; Python; Ruby; JavaScript; types inconsistency; defects traces.

References

[1]. L. D. Paulson. Developers shift to dynamic programming languages. IEEE Comp vol. 40, issue 2, February 2007, pp. 12–15.
[2]. I. E. Bronstein Type inferrence for Python prigramming language. rudy ISP RAN [The Proceedings of ISP RAS], 2013, vol 24, pp. 161-190 (in Russian)
[3]. TIOBE Programming Community Index for August 2013: http://tinyurl.com/tiobe-201308
[4]. V. O. Savitsky, D. V. Sidorov, Inkremental'nyj analiz iskhodnogo koda na yazykakh C/C++[Incremental source code analysis for C/C++ languages] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 22, pp. 119-129 (in Russian)
[5]. A. Avetisyan, A. Belevantsev, A. Borodin, V. Nesov Ispol'zovanie staticheskogo analiza dlya poiska uyazvimostej i kriticheskikh oshibok v iskhodnom kode programm[Using static analysis for finding security vulnerabilities and critical errors in source code] Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol 21, pp. 23-38 (in Russian)
[6]. Pyflakes: https://launchpad.net/pyflakes
[7]. Pylint - code analysis for Python: http://www.pylint.org
[8]. reek: https://github.com/troessner/reek
[9]. ruby-lint: http://code.yorickpeterse.com/ruby-lint/latest
[10]. JSLint: http://www.jslint.com
[11]. JSHint, a JavaScript Code Quality Tool: http://www.jshint.com
[12]. PEP 8 – Style Guide for Python Code: http://www.python.org/dev/peps/pep-000
[13]. R. Chillarege, I. Bhandari, J. Chaar, M. Halliday, D. Moebus, B. Ray, M. Y. Wo Orthogonal Defect Classification — A Concept of In-Process Measurements (19 [14] W. Humprey. A discipline for software engineering (1997).
[14]. W. Humprey. A discipline for software engineering (1997).
[15]. M. Reingart. Rad2py: Plataforma de Trabajo para el Desarrollo Rápido de Aplicaciones bajo un Proceso de Software Personal, pp. 152–153: http://code.google.com/p/rad2py/wiki/DefectTypeStandard

[16]. Bugs : Exaile: https://bugs.launchpad.net/exaile
[17]. Bugs : calibre: https://bugs.launchpad.net/calibre
[18]. sqlalchemy: http://www.sqlalchemy.org/trac
[19]. Bugs : Bazaar: https://bugs.launchpad.net/bzr
[20]. Twisted: http://twistedmatrix.com/trac
[21]. The Trac Project: http://trac.edgewall.org
[22]. Django: https://code.djangoproject.com
[23]. DRuby - Home: http://www.cs.umd.edu/projects/PL/druby
[24]. Type Analysis for JavaScript: http://www.brics.dk/TAJS
[25]. Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, Michael Hicks. Static type inference for Ruby. SAC '09 Proceedings of the 2009 ACM symposium on Applied Computing, pp. 1859–1866.
[26]. jruby-parser: https://github.com/jruby/jruby-parser
[27]. esprima: https://github.com/ariya/esprima
[28]. tirpan: https://github.com/bronikkk/tirpan
[29]. Gramps Bug Report 005023: http://www.gramps-project.org/bugs/view.php?id=5023

Моделирование окружения драйверов устройств операционной системы Linux[1]

Захаров И.С., Мутилин В.С., Новиков Е.М., Хорошилов А.В.
{ilja.zakharov, mutilin, novikov, khoroshilov}@ispras.ru

Аннотация. При статической верификации драйверов устройств операционной системы Linux необходимо учитывать особенности взаимодействия драйверов с сердцевиной ядра, так как это взаимодействие оказывает определяющее влияние на работу драйвера. В то же время, верификации драйвера в комбинации с исходным кодом сердцевины ядра не представляется возможной ввиду сложности и объема получающегося кода. В качестве решения этой проблемы в статье предлагается метод моделирования окружения драйверов на основе π-исчисления Р.Милнера и метод трансляции π-модели окружения в программу на языке Си, которая при связывании с исходным кодом драйвера описывает с точки зрения инструментов статической верификации те же сценарии работы драйвера, что и реальное окружение драйвера в операционной системе.

Ключевые слова: операционная система; драйвер; окружение; верификация.

1 Введение

Обеспечение надежности и информационной безопасности программных и программно-аппаратных систем является одной из главных задач современных информационных технологий. Особый вклад в решение этой задачи вносят работы по верификации системного программного обеспечения, в первую очередь компонентов операционных систем (ОС). Верификация драйверов ОС Linux является критически важной задачей, так как:

- Корректность драйверов является необходимым условием обеспечения надежности и безопасности систем, поскольку драйверы работают с тем же уровнем привилегий, что и остальное ядро [1].

[1] Работа поддержана ФЦП "Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы" (контракт N 11.519.11.4006)

- Драйверы ОС Linux, входящие в ядро, это большой, стремительно растущий класс программных систем. На данный момент суммарный объем исходного кода драйверов составляет более 9 миллионов строк. За последний год он увеличился на полмиллиона строк.

1.1. Драйверы операционной системы Linux

В ядре Linux можно выделить «сердцевину» ядра и драйверы (см. Рис. 1). Сердцевина ядра отвечает за управление процессами и памятью, содержит сетевую подсистему и др. Драйверы позволяют ОС и пользовательским приложениям использовать возможности соответствующей аппаратуры. Они составляют большую часть исходного кода ядра – около 70%. Большинство драйверов могут быть скомпилированы как динамически загружаемые модули ядра Linux.

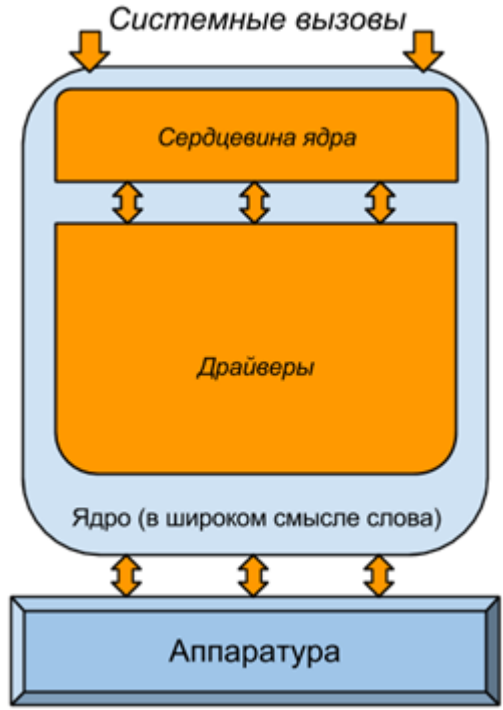


Рис. 1. Драйверы в ядре ОС Linux

Драйверы сильно отличаются от традиционных программ на языке Си: они не имеют функции *main*, а последовательность выполнения кода во многом зависит от взаимодействия драйверов с сердцевиной ядра. Особенности устройства драйвера рассмотрены ниже на примере драйвера USB CDC Phonet

(*cdc-phonet.c*) из сетевой USB подсистемы (*drivers/net/usb*) ядра версии 3.0.1 (см. Рис. 2):

- **Функция инициализации** (далее функция *init*). Драйвер начинает свою работу после загрузки соответствующего модуля в память. Загрузка модуля происходит в процессе старта ОС Linux или после того, как в процессе работы возникла необходимость использовать соответствующее устройство. В процессе загрузки модуля драйвера сердцевина ядра вызывает функцию инициализации драйвера. В примере на Рис. 2 такой функцией является *usbpn_init*. Если инициализация проходит успешно, то возвращается значение «0», в противном случае возвращается соответствующий код ошибки.
- **Функция выхода** (далее функция *exit*). Устройство можно использовать до тех пор, пока соответствующий модуль не будет выгружен из памяти. Перед выгрузкой сердцевина ядра выполняет специальную функцию драйвера *exit*, в которой, например, освобождаются захваченные драйвером ресурсы. На Рис. 2 такой функцией является *usbpn_exit*.

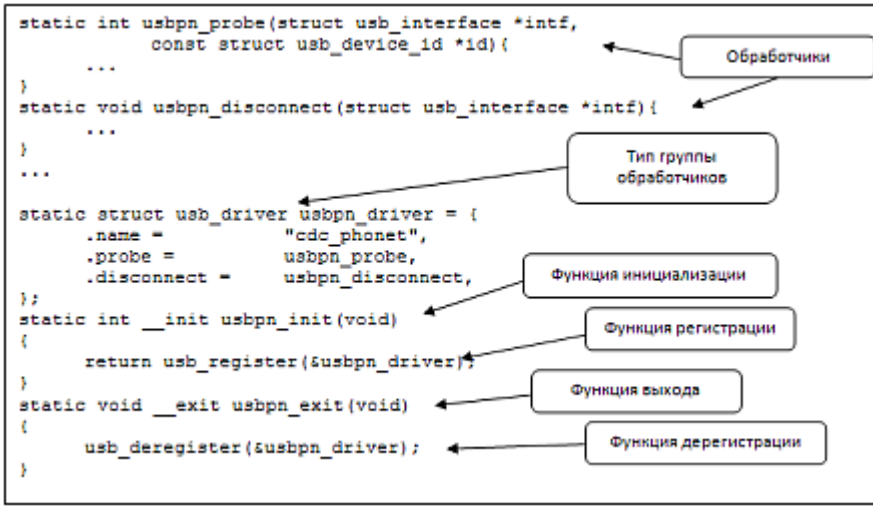


Рис. 2. Выдержки из исходного кода драйвера drivers/net/usb/cdc-phonet.c

- **Функции-обработчики.** Основная функциональность драйвера сосредоточена в функциях-обработчиках событий (далее *обработчики*). Обработчики вызываются сердцевиной ядра, когда требуется обработать события, связанные с драйвером, например, прерывания от устройств или системные вызовы от пользовательских приложений. В примере на Рис. 2 обработчиками являются функции *usbpn_probe* и *usbpn_disconnect*.

- **Группы обработчиков.** Большинство обработчиков объединены в группы. Каждая группа имеет определенный тип в зависимости от назначения ее обработчиков. Порядок вызова обработчиков и параметры их вызова сердцевиной ядра определяются ролями обработчиков в группе. В большинстве случаев, тип группы обработчиков можно определить по типу структуры, в полях которой хранятся функциональные указатели на обработчики (например, на Рис. 2 *usb_driver*). Поля структуры определяют роли, которые выполняют хранящиеся в этих полях указатели на функции-обработчики в конкретном драйвере. Для работы с *USB* устройством в примере на Рис. 2 представлена группа из двух обработчиков *usbpn_probe* и *usbpn_disconnect* с типом *usb_driver*. Данная группа задана при помощи структуры *struct usb_driver* с полями *probe* и *disconnect*.
- **Использование библиотечных функций сердцевины ядра.** Так как сердцевина ядра может вызывать только зарегистрированные обработчики, то наиболее важными библиотечными функциями с точки зрения моделирования окружения являются библиотечные функции регистрации и дерегистрации групп обработчиков драйвера. В начале некоторые обработчики регистрируются в процессе выполнения функции инициализации модуля, остальные регистрируются при вызове обработчиков, зарегистрированных ранее. Дерегистрация групп обработчиков часто осуществляется во время выполнения функции выхода драйвера. В примере на Рис. 2 группа обработчиков *usb_driver* регистрируется при помощи функции *usb_register* и дерегистрируется с помощью функции *usb_deregister*, описанных в заголовочном файле ядра *include/linux/usb.h*. Данные функции вызываются в теле функций инициализации и выхода драйвера. Не все обработчики можно объединить в группы, например обработчики прерываний или функции выполнения отложенных задач в очередях, таймерах, тасклетах и т.д. Подобные обработчики также требуют соответствующей регистрации и дерегистрации и вызываются сердцевиной ядра.

Рассмотренный пример демонстрирует, что работа драйвера существенным образом зависит от внешних событий в системе, которые приводят к вызовам сердцевиной ядра функций инициализации, выхода и функций-обработчиков. Драйвер, в свою очередь, использует библиотечные функции ядра, такие как функции регистрации и дерегистрации.

1.2. Статическая верификация драйверов

Осуществлять поддержку корректности драйверов ОС Linux вручную слишком трудоемкая задача. Одним из направлений ее решения является применение инструментов статической верификации. Работа драйвера

неотделимо связана с работой сердцевины ядра, но анализировать модуль драйвера вместе с ней слишком сложная задача для инструментов верификации на сегодняшний день. Причиной этому являются высокая сложность и большой объем исходного кода сердцевины ядра. Для решения этой проблемы драйвер нужно заключить в некоторое «окружение» с тем, чтобы программный код был замкнутым. Такое окружение будем называть *синтезируемым окружением*. К синтезируемому окружению предъявляются две группы требований.

Первая группа требований определяет ограничения на конструкцию (структуру) синтезируемого окружения, обусловленные возможностями современных инструментов статической верификации. Так, большинство из них работает с последовательными программами на языке программирования Си, имеющими точку входа (аналог функции *main*), с которой начинается выполнение. Кроме того, в значительной части имеются ограничения по точности анализа функциональных указателей, рекурсивных структур данных и т.д.

Наибольший интерес представляет вторая группа требований к окружению, связанная с тем, что оно должно воспроизводить те же сценарии взаимодействия с драйвером, что и реальное окружение драйвера в операционной системе. Чтобы не упустить ошибки, окружение должно быть полным, т.е. включать все последовательности событий, которые могут происходить в реальной системе с учетом всевозможных событий со стороны пользователей и аппаратуры. При этом окружение должно быть корректным.

Основными видами требований корректности к синтезируемому окружению являются:

1. Требования к порядку вызова функций *init* и *exit*.
2. Требования к вызову обработчиков, принадлежащих одной группе, включающие:
    а) Ограничения на параметры вызова обработчиков;
    б) Ограничения на порядок вызова обработчиков.
3. Требования к порядку и к параметрам вызова обработчиков из разных групп обработчиков.
4. Требования к моделированию контекста вызова обработчиков (учет возможности вызова прерываний, учет захваченных сердцевиной ядра блокировок и др.).

Нарушение требований корректности может приводить к ложным сообщениям об ошибках, т.е. ситуациям, когда инструмент статической верификации сигнализирует об ошибке, но ошибка в реальном окружении не воспроизводима.

Примером требований к порядку вызовов в рамках одной группы обработчиков типа *usb_driver* является ограничение на порядок вызовов функций с ролями *probe* и *disconnect*. Функция с ролью *disconnect* может быть вызвана, только если до этого был сделан успешный вызов функции с ролью

probe. Также при вызове *disconnect* должен передаваться тот же указатель на структуру *usb_interface*, которая была успешно инициализирована функцией *probe*. Если в группу типа *usb_driver* входят функции с ролями *suspend, resume*, то их вызов возможен только после успешного вызова *probe* и до вызова *disconnect*. Другой пример: перед вызовом некоторых функций требуется, чтобы был захвачен мьютекс.

В качестве примера требований на порядок вызова обработчиков из разных групп, возьмем группы обработчиков типа *usb_driver* и *file_operations*. Вызов функций из группы типа *file_operations* возможен, только после успешной регистрации данной группы в обработчике с ролью *probe* из группы типа *usb_driver*.

Другой пример требований: после успешного вызова *probe* окружение не должно вызывать функцию выхода *exit* до вызова *disconnect*.

1.3. Моделирование окружения драйверов

Окружение моделирует работу реальной сердцевины ядра, которая получает события как из пространства пользователя в виде системных вызовов, так и от аппаратуры. В зависимости от приходящих событий, окружение выбирает один из доступных обработчиков драйвера и вызывает его, а кроме того, изменяет свое состояние. В процессе выполнения обработчика, драйвер вызывает библиотечные функции окружения, например, вызывает функции регистрации новых групп обработчиков, которые становятся доступными для вызова из окружения.

Окружение и драйвер можно рассматривать как процессы в π-исчислении Р.Милнера [7]. Процесс выполняет шаги, каждый из которых – это прием или посылка сигнала и переход к следующему процессу. Вызов обработчиков драйвера можно рассматривать как посылку сигнала от окружения к драйверу, а возвраты значений как посылку сигнала от драйвера к окружению. Получая сигнал, драйвер осуществляет его обработку, при этом он может обращаться к библиотечным функциям сердцевины ядра, что также можно рассматривать как посылку и прием сигналов.

Переход к следующему процессу моделирует изменение состояния, в частности за счет этого моделируется порядок вызова обработчиков. Например, в реальном окружении до вызова обработчика с ролью *probe*, оно находится в состоянии, в котором нельзя вызывать обработчик с ролью *disconnect*. Это моделируется процессом, принимающим только сигнал *probe*. Как только окружение получит от драйвера сигнал с успешным кодом возврата из *probe*, обработчик с ролью *disconnect* можно будет вызывать. Это моделируется переходом по приему сигнала успешного возврата к следующему процессу, в котором доступен сигнал *disconnect*. Тем самым обеспечивается порядок вызова обработчиков с ролями *probe* и *disconnect*.

Предлагаемый в данной статье метод построения синтезируемого окружения состоит из двух шагов (см. Рис. 3). На первом шаге строится π-модель реального окружения. Эта модель учитывает требования полноты и корректности, накладываемые на окружение. Для формального описания модели используется π-исчисление Р.Милнера [7]. С помощью данного исчисления удается довольно компактно и наглядно описывать последовательности вызовов обработчиков драйвера (сценарии воздействия на драйвер). Кроме того, π-исчисление позволяет учитывать многопоточность окружения и асинхронность взаимодействий драйвера с окружением.

На втором шаге из π-модели окружения генерируется код на языке Си. Для исходного кода драйвера генерируется обертка, которая преобразует сигналы π-модели в вызовы обработчиков драйвера так, чтобы драйвер соответствовал своей π-модели. В результате на выходе имеется программа, которая соответствует π-модели, и при этом является однопоточной и отвечает ограничениям инструментов верификации.

Поскольку трансформация многопоточной π-модели в однопоточную Си программу не является тривиальной, возникает необходимость доказать корректность построения синтезируемого окружения. Дается определение эквивалентности между моделью в π-исчислении и программой на языке Си, в котором учитываются трассы (последовательности событий), содержащие только общие действия для процессов драйвера и окружения с ограничением на переключения при выполнении обработчиков драйвера. Далее приводится теорема о том, что для каждого драйвера в предположении, что код драйвера вместе с обертками эквивалентен его π-модели, π-модель эквивалентна синтезированной Си программе.

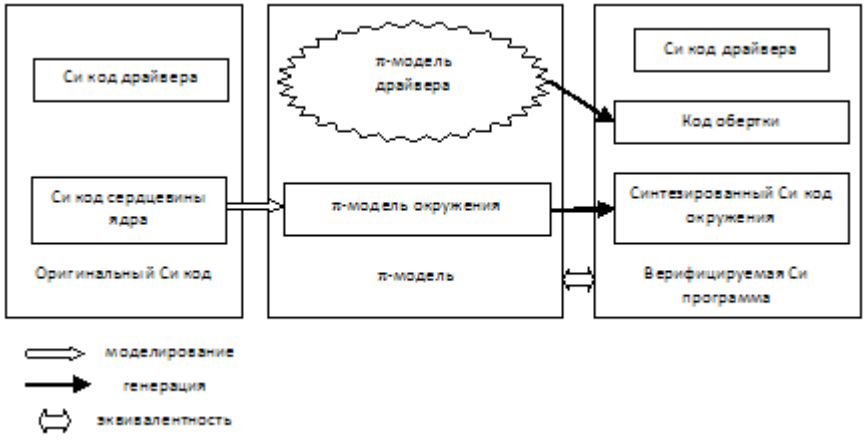


Рис. 3. Моделирование окружения драйверов

2. Основные определения

Введем понятия, используемые в π-исчислении. Синтаксис π-исчисления позволяет представлять *процессы* – абстракции независимых потоков управления. *Исчисление* позволяет задавать параллельную композицию процессов, синхронное взаимодействие между процессами с помощью сигналов, создание процессов и недетерминизм. Сигнал задается *меткой* (*каналом*) – абстракция связи между двумя процессами, с помощью которой осуществляется взаимодействие.

Сигналы могут иметь параметры. В определении «The Polyadic π-Calculus: a Tutorial, Robin Milner, October 1991» [7] допускается передача любых меток в качестве параметра сигнала.

Множество меток приема сигналов обозначим как A. Множество меток посылки сигнала обозначим как $\bar{A}$, оно состоит из меток $\bar{\alpha}$, где $\alpha \in A$. Обозначим $\Lambda = A \cup \bar{A}$.

Множество действий включает в себя действия посылки/получения для меток, а также специальное пустое действие τ, $Act = \Lambda \cup \tau$.

Непустые действия могут иметь параметры, которые записываются в круглых скобках. Жирным шрифтом будем обозначать вектор параметров, например $\boldsymbol{x}$.

Для действий посылки сигнала $\overline{\alpha(e(\boldsymbol{x}))}$ вычисляется значение выражения $e(\boldsymbol{x})$, на которое заменяется переменная $\boldsymbol{y}$ в действии приема сигнала $\alpha(\boldsymbol{y})$.

Кроме того, в определении для удобства записи добавлен оператор *if-then-else*.

Определение 1.

Процессы верхнего уровня:

$$P := P_1 \mid P_2 \mid !P \mid (\nu\alpha)P \mid N$$

где

- $P_1 \mid P_2$ – параллельная композиция, одновременно выполняются процессы P_1, P_2;
- $(\nu\alpha)$P – создание новой метки $\alpha \in A$;
- $!P$ – создание копии процесса;
- N – процесс нижнего уровня.

Процессы нижнего уровня:

$$N(\boldsymbol{x}) := 0 \mid K_1 + \cdots + K_n \mid \; if\; b(\boldsymbol{x})\; then\; N_1(\boldsymbol{x})\; else\; N_2(\boldsymbol{x})$$

- 0 – пустой процесс;
- $K_1 + \cdots + K_n$ – выбор, где процесс может продолжаться одним из вариантов:
    - $N_i = \alpha_i(y_i).K_i(e_i(\boldsymbol{x}, y_i))$, $\alpha_i \in \Lambda$ – прием сигнала;

- $N_i = \overline{\alpha_i(e(\boldsymbol{x}))}.K_i(e_i(\boldsymbol{x}))$ – посылка сигнала;
- $N_i = K_i(e_i(\boldsymbol{x}))$ – τ действие;

- $if\ b(\boldsymbol{x})\ then\ N_1(\boldsymbol{x})\ else\ N_2(\boldsymbol{x})$ – если условие $b(\boldsymbol{x})$ выполнено, то процесс продолжается как $N_1(\boldsymbol{x})$, иначе как $N_2(\boldsymbol{x})$.

Для сокращенного задания идентификаторов процессов нижнего уровня будем использовать запись:

$$N_j(\boldsymbol{x}) := \cdots.\langle N_i\rangle \text{ выражение1 ...}$$

которая означает:

$$N_j(\boldsymbol{x}) := \cdots.N_i(\boldsymbol{x}) \ldots$$

$$N_i(\boldsymbol{x}) := \text{ выражение1}$$

Как можно видеть, в определении на верхнем уровне предусмотрены конструкции порождения новых процессов, создание новых меток. Процессы верхнего уровня могут быть запущены параллельно.

Для каждого процесса верхнего уровня определяются процессы нижнего уровня. Так как параллельная композиция определена для процессов верхнего уровня, то процессы нижнего уровня можно рассматривать как состояния соответствующего процесса верхнего уровня. Осуществляя переходы от одного процесса нижнего уровня к следующему процессу, процесс верхнего уровня меняет свое состояние.

Конструкция выбора $K_1 + \cdots + K_n$ позволяет задавать множество доступных для посылки сигналов и множество сигналов, которые процесс готов принять. При выборе одного из действий процесс переходит в следующее состояние, тем самым меняются множества доступных сигналов. Это позволяет моделировать различные ограничения на порядок вызова обработчиков. Моделируется это с помощью процессов нижнего уровня следующим образом. При получении сигнала вызова, для которого нужно наложить ограничение на порядок вызова, мы переходим к следующему процессу, в котором доступны уже другие сигналы в соответствии с допустимыми порядками вызова. Например, как только мы получили сигнал *probe* в некотором процессе N_1, он переходит к процессу N_2, в котором множество доступных действий включает посылку сигнала *disconnect*.

3. Формальная модель драйвера и его окружения в π-исчислении

Рассмотрим моделирование окружения с помощью π-исчисления. При моделировании драйвер рассматривается как набор процессов, запущенных параллельно:

$$Drv_\pi \coloneqq P_{init} \mid P_{exit} \mid !\, P_{fcall}$$

Процессы P_{init} и P_{exit} представляют функции инициализации *init* и *exit* соответственно. Эти процессы принимают на вход сигналы *init(ret)*, *exit(ret)* и отвечают на них $\overline{ret(x)}$, $\overline{ret()}$ соответственно. Процесс P_{fcall} представляет обработчики драйвера, которые окружение может вызывать после регистрации. Эти процессы принимают сигнал вызова функции обработчика $\overline{f(ret_i, f_i, ctx_i, params_i)}$, в параметрах сигнала которого передаются: ret_i – метка сигнала для ответа, f_i – вызываемая функция, ctx_i – контекст вызова, $params_i$ – параметры вызова. Причем, так как обработчики могут выполняться параллельно, для каждого вызова порождается отдельная копия процесса. В ответ P_{fcall} посылает сигнал с возвращаемым значением $ret_i(result_i)$.

Во время выполнения инициализации, выхода, а также при выполнении других обработчиков драйвер может:

- обращаться к библиотечным функциям сердцевины ядра $\overline{g_1(ret_1, params_1)}, \dots, \overline{g_l(ret_l, params_l)}$ и получать от них ответы $ret_1(result_1), \dots, ret_l(result_l)$,
- обращаться к глобальным переменным, с помощью сигналов $\overline{set_{v_i}(x)}, get_{v_i}(x)$ к процессам P_{v_i}.

Таким образом, π-модель окружения предоставляет обработку:

- вызовов библиотечных функций $g_1(ret_1, params_1), \dots, g_l(ret_l, params_l)$;
- обращений к глобальным переменным, с помощью сигналов $\overline{set_{v_i}(x)}$, $get_{v_i}(x)$.

А также осуществляет вызовы:

- функций инициализации и выхода драйвера $\overline{init(ret)}$, $\overline{exit(ret)}$;
- функций-обработчиков для зарегистрированных групп $\overline{f(ret_i, f_i, ctx_i, params_i)}$.

В самый начальный момент времени активируется основной процесс окружения P_{module}, отвечающий за инициализацию и выход. Рассмотрим пример главного процесса окружения P_{module}, который вызывает функции инициализации и выгрузки драйвера:

$P_{module} \coloneqq L0$

$L0 \coloneqq (vret)L1$

Процесс *L1* посылает драйверу сигнал инициализации *init*. Драйвер осуществляет инициализацию, посылая сигналы сердцевине ядра. Например, для выделения необходимых ресурсов и регистрации группы функций обработчиков драйвера. Если она прошла успешно, то посылается сигнал ret с кодом ответа 0, в противном случае, посылается соответствующий код ошибки:

$L1 \coloneqq \overline{init(ret)}.\langle L3\rangle ret(r).\langle L4\rangle if\ r == 0\ then\ L2\ else\ 0,$

где у процесса *L4* имеется параметр *r*, через который передается возвращаемое значение функции *init*.

Окружение взаимодействует с процессом драйвера до тех пор, пока не будет послан сигнал *exit*. Обрабатывая данный сигнал, драйвер посылает сигналы окружению, например, для освобождения занятых ресурсов и дерегистрации групп обработчиков.

$L2 \coloneqq \overline{mstop}.\langle L5\rangle\overline{exit(ret)}.\langle L6\rangle ret.0$

Процесс P_{module} обращается к процессу $P_{trymoduleget}$, который моделирует захват модуля драйвера, так чтобы его нельзя было выгрузить, т.е. окружение не вызвало функцию выхода *exit*. Процесс, захватывающий модуль драйвера, посылает сигнал $\overline{tmg}$, и получает ответ $tmg^{ret}(true)$ в случае успешного захвата модуля. Сигнал *mstop* нужен для того, чтобы процесс не допускал новые захваты.

$P_{trymoduleget} \coloneqq M(0)$

$M(0) \coloneqq tmg.\langle M1\rangle\overline{tmg^{ret}(true)}.M(1) + mstop.MD$

Для $i \geq 1$:

$M(i) \coloneqq tmg.\langle M2\rangle\overline{tmg^{ret}(true)}.M(i+1) + mput.\langle M3\rangle\overline{mput^{ret}}.M(i-1)$

$MD \coloneqq mstop.MD + tmg.\langle MD1\rangle\overline{tmg^{ret}(false)}.MD$

На Рис. 4 показаны изображения процессов P_{module} и $P_{trymoduleget}$ в виде графов. В вершинах изображены состояния процессов нижнего уровня, на дугах показаны события посылки и приема сигналов. Процесс $P_{trymoduleget}$ имеет потенциально неограниченное число состояний соответствующих параметру *i*.

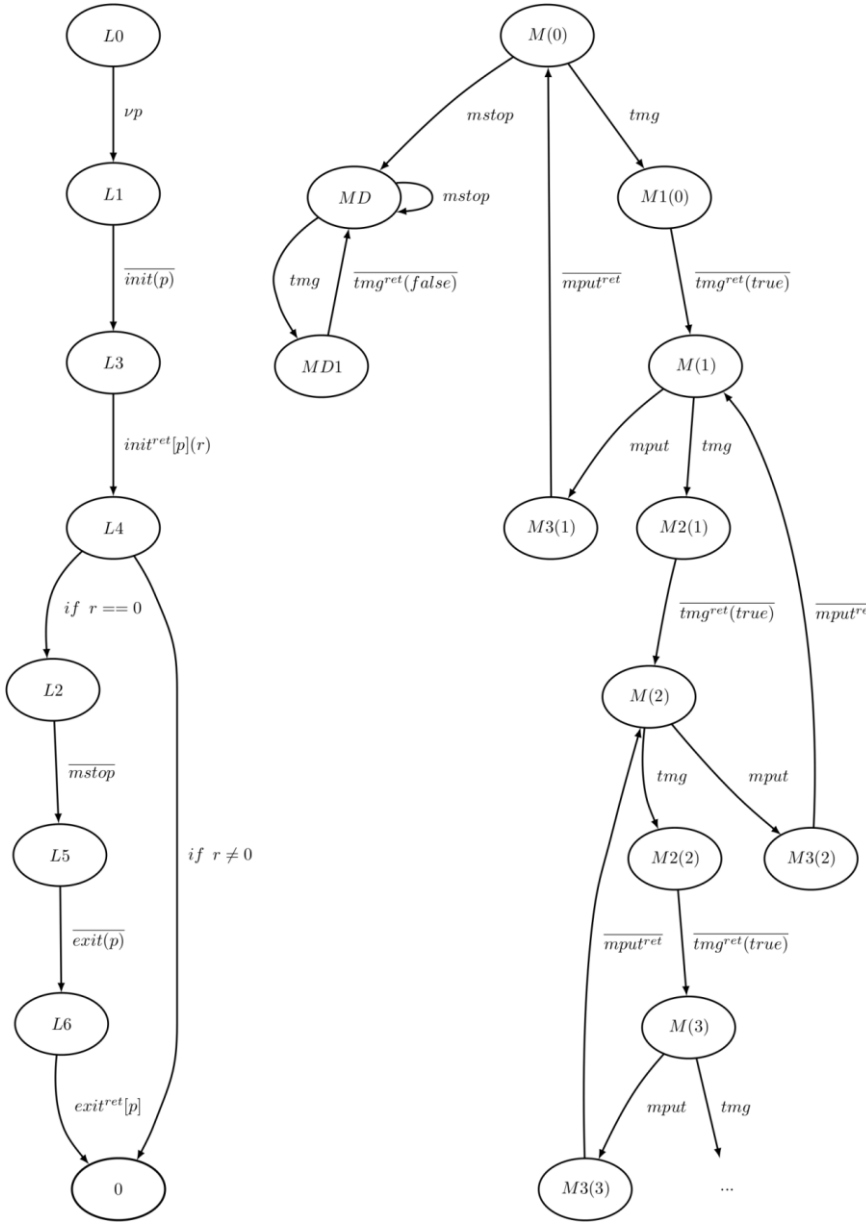


Рис. 4. Пример процесса P_{module} (слева) и процесса $P_{trymoduleget}$ (справа)

Регистрация групп обработчиков осуществляется с помощью библиотечных функций сердцевины ядра. В библиотечные функции передается группа обработчиков, как правило, в виде структуры языка Си из указателей на функции обработчики.

В примере драйвера на Рис. 2 вызывается функция-регистрации *usb_register*, которой передается указатель на переменную *usbpn_driver* типа *usb_driver*, содержащую обработчики *probe* и *disconnect*.

Группа обработчиков работает с некоторыми сущностями ядра, и в зависимости от этого набора сущностей создается соответствующий набор экземпляров процессов группы обработчиков. Например, для группы обработчиков *file_operations*, предназначенной для работы с файлами, такой сущностью является файл, который может быть в состояниях открыт и закрыт. В открытом состоянии хранится текущее смещение внутри файла, используемое операциями чтения и записи.

Для того чтобы моделировать порождение сущностей, с которыми осуществляются операции вводится специальный процесс регистрации. Для каждой сущности он создает процесс группы, который осуществляет вызовы обработчиков из группы с данной сущностью. Если известно для скольких сущностей предназначена группа обработчиков, то создается соответствующее количество экземпляров процесса группы. Неограниченное число экземпляров может моделироваться как недетерминированное порождение новых сущностей. Если имеется единственная сущность, то создается единственный процесс группы.

Рассмотрим пример моделирования регистрации для группы обработчиков типа *usb_driver*. В этом примере для группы будем создавать единственный экземпляр P_{usb_driver} процесса для каждой регистрируемой группы, поэтому специального процесса регистрации не требуется.

$$P_{usb\_driver} \coloneqq S0$$

$$S0 \coloneqq\ !\,usb\_register(usb\_deregister, probe, disconnect).S1$$

Для краткости далее параметры *probe* и *disconnect* опускаются.

$$S1 \coloneqq\ usb\_deregister.0\ +\ \overline{tmg}.\langle S4\rangle tmg^{ret}(r).\langle S5\rangle if\ r\ then\ S2\ else\ S1$$

$$S2 \coloneqq\ \overline{f(ret, probe)}.\langle S6\rangle ret(r).$$

$$\langle S7\rangle\ if\ r == 0\ then\ S3\ else\ S2\ +\ usb\_deregister.\langle S8\rangle\overline{mput}.0$$

$$S3 \coloneqq\ \overline{f(ret, disconnect)}.\langle S9\rangle ret.S2$$

Для моделирования зависимостей, как между группами, так и зависимостей с функциями инициализации и выхода используются сигналы между процессами. В примере на Рис.2 модуль драйвера нельзя выгружать после того как процесс перешел в состояние $S2$, в котором он готов вызывать

обработчики *probe* и *disconnect*. С точки зрения окружения, это означает, что нельзя вызывать функцию выхода *exit* в основном процессе окружения P_{module}. Для хранения состояния счетчика захватов модуля используется процесс $P_{trymoduleget}$.

Поэтому перед тем, как вызывать какие либо обработчики, процесс группы типа *usb_driver* посылает сигнал $\overline{tmg}$. В случае успеха, он переходит в состояние $S2$. В случае, если начата выгрузка модуля, то процесс остается в состоянии $S1$ и ожидает дерегистрацию, тем самым не осуществляя вызовы обработчиков группы usb_driver.

Другой пример взаимодействия процессов можно продемонстрировать на примере группы *file_operations*. Как только файл переходит в открытое состояние после успешного вызова *open*, необходимо обеспечить невозможность выгрузки модуля. Для этого также используется процесс $P_{trymoduleget}$.

Ограничения на порядок вызова моделируются с помощью процессов S1, …, S9. Например, для установления порядка между *probe* и *disconnect*, как только мы получили сигнал *probe* в процессе $S2$, он переходит к процессу $S3$, в котором множество доступных действий включает посылку сигнала *disconnect*.

Требования к порядку вызова обработчиков нескольких групп учитываются за счет передачи сообщений между процессами групп.

4. Трансляция процессов в Си-программу

Трансляция процессов описана в приложении А диссертации [8]. В разделе А.1 описан общий метод трансляции в многопоточную программу, а в разделе А.2 описывается его модификация для трансляции в последовательную Си программу.

Для упрощения трансляции в диссертации [8] используется модифицированное определение π-процессов. Вводится разделение на статические и динамические метки. Статические – это метки, которые не передаются в качестве параметров сигнала. Данные метки были введены, так как для них можно упростить генерируемый код.

Вместо того чтобы передавать непосредственно сами метки в качестве параметра, вводятся и передаются специальные параметры меток со значениями из потенциально бесконечного множества П. Метки, которые имеют специальные параметры, называются динамическими, обозначим $\Phi = F \cup \bar{F}$, где F – метки приема и $\bar{F}$ – метки посылки. Специальные параметры метки пишутся в квадратных скобках после метки.

Динамически связываемые метки приема сигнала $f[p_1]$ и посылки сигнала $\bar{f}[p_2]$ осуществляют взаимодействие, только если значения параметров меток совпадают. Т.е. процесс получатель принимает сигнал $f[p_1]$, только если

значение передаваемого параметра метки p_1 совпадает со значением получателя p_2. Процесс отправитель считает сигнал $\bar{f}[p_2]$ отправленным, только если нашелся получатель с совпадающим значением параметра метки p_1.

Отметим, что введение динамических меток необходимо нам лишь для удобства записи трансляции в Си программу. Любую модель, записанную в классических π-процессах, можно свести к π-процессам с динамическими метками. Метки, не передаваемые как параметры, записываются как статические. Каждой метке, передаваемой через параметры, ставим в соответствие натуральное число $p \in \Pi$. Вводим динамическую метку *f*. Заменяем все места, в которых использовались метки для приема и посылки на динамическую метку *f[p]* с соответствующим параметром *p*. Там, где метка передавалась в качестве параметра, передаем значение *p*.

Например, если имеются процессы:

$P := \nu x\ \overline{a(x)}.x()$

$Q := a(x).\overline{x()}$

то их можно переписать как:

$P := \nu p\ \overline{a(p)}.f[p]()$

$Q := a(p).\overline{f[p]()}$

4.1. Отношение редукции

При первом прочтении можно пропустить определение отношения редукции, которая требуется для задания формальной семантики π-процессов с динамическими метками и формулировки теоремы.

Вспомогательное отношение структурной эквивалентности ≡ определяется аналогично [7].

Определим отношение редукции → над процессами с динамическими метками аналогично [7], при этом пометим пары в отношении метками α из $A \cup F \cup \{\tau\}$, т.е. метками приема сигналов и пустой меткой τ.

Отношение редукции $\xrightarrow{\alpha}$ над процессами – это наименьшее отношение, удовлетворяющее следующим правилам:

- Если есть определение процессов $K(\boldsymbol{x})$ и $N(\boldsymbol{x})$, в каждом из которых есть слагаемое, которое может взаимодействовать с другим, то это взаимодействие осуществляется с меткой приема и значениями

переданных параметров.

$$COMM_1(\alpha \in A): \frac{\begin{array}{c} K(\boldsymbol{x}) \coloneqq \left(\cdots + \alpha(y).K_i\left(e_i(\boldsymbol{x},y)\right)\right) \\ N(\boldsymbol{x}) \coloneqq \left(\cdots + \overline{\alpha(e(\boldsymbol{x}))}.N_j\left(e_j(\boldsymbol{x})\right)\right) \end{array}}{K(\boldsymbol{u}) \mid N(\boldsymbol{v}) \xrightarrow{\alpha(e(\boldsymbol{v}))} K_i\left(e_i(\boldsymbol{u},\ e(\boldsymbol{v}))\right) \mid N_j(e_j(\boldsymbol{v}))}$$

- Для динамических меток сравниваются также значения параметров ($\boldsymbol{x}_i$ – i-ый элемент вектора $\boldsymbol{x}$).

$$COMM_2(\alpha \in F): \frac{\begin{array}{c} K(\boldsymbol{x}) \coloneqq \left(\cdots + \alpha[\boldsymbol{x}_i](y).K_i\left(e_i(\boldsymbol{x},y)\right)\right) \\ N(\boldsymbol{x}) \coloneqq \left(\cdots + \overline{\alpha[\boldsymbol{x}_i](e(\boldsymbol{x}))}.N_j\left(e_j(\boldsymbol{x})\right)\right) \\ \boldsymbol{u}_i = \boldsymbol{v}_j \end{array}}{K(\boldsymbol{u}) \mid N(\boldsymbol{v}) \xrightarrow{\alpha[\boldsymbol{u}_i](e(\boldsymbol{v}))} K_i\left(e_i(\boldsymbol{u},\ e(\boldsymbol{v}))\right) \mid N_j(e_j(\boldsymbol{v}))}$$

- Переход по τ действию может осуществляться всегда.

$$TAU: \frac{K(\boldsymbol{x}) \coloneqq \left(\cdots + \tau.K_i\left(e_i(\boldsymbol{x})\right)\right)}{K(\boldsymbol{u}) \xrightarrow{\tau} K_i\left(e_i(\boldsymbol{u})\right)}$$

- if-then-else оператор.

$$IFTHEN: \frac{\begin{array}{c} K(\boldsymbol{x}) \coloneqq \boldsymbol{if}\ b(\boldsymbol{x})\ \boldsymbol{then}\ K_1(\boldsymbol{x})\ \boldsymbol{else}\ K_2(\boldsymbol{x}) \\ b(\boldsymbol{u}) = TRUE \end{array}}{K(\boldsymbol{u}) \xrightarrow{\tau} K_1(\boldsymbol{u})}$$

$$IFELSE: \frac{\begin{array}{c} K(\boldsymbol{x}) \coloneqq \boldsymbol{if}\ b(\boldsymbol{x})\ \boldsymbol{then}\ K_1(\boldsymbol{x})\ \boldsymbol{else}\ K_2(\boldsymbol{x}) \\ b(\boldsymbol{u}) = FALSE \end{array}}{K(\boldsymbol{u}) \xrightarrow{\tau} K_2(\boldsymbol{u})}$$

- В параллельной композиции редукция компонентов может осуществляться независимо.

$$PAR: \frac{K(\boldsymbol{u_1}) \xrightarrow{\alpha} K'(\boldsymbol{u_2})}{K(\boldsymbol{u_1}) \mid N(\boldsymbol{v}) \xrightarrow{\alpha} K'(\boldsymbol{u_2}) \mid N(\boldsymbol{v})}$$

- Создание нового значения.

$$RES: \frac{K(\boldsymbol{u_1}) \xrightarrow{\alpha} K'(\boldsymbol{u_2})}{(\nu x)K(\boldsymbol{u_1}) \xrightarrow{\alpha} (\nu x)K'(\boldsymbol{u_2})}$$

- Редукция для структурно эквивалентных процессов.

$$STRUCT: \frac{K \equiv N; N(\boldsymbol{u_1}) \xrightarrow{\alpha} N'(\boldsymbol{u_2}); N' \equiv K'}{K(\boldsymbol{u_1}) \xrightarrow{\alpha} K'(\boldsymbol{u_2})}$$

Понятие редукции позволяет нам определить множество возможных трасс выполнения π процесса P, обозначаемое как $traces_\pi(P)$. Для редукции

$P \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} P_2 \cdots \xrightarrow{\alpha_n} P_n$, определим трассу как подпоследовательность $\alpha_{i_1}, \ldots, \alpha_{i_k}$ последовательности $\alpha_1, \ldots, \alpha_n$, включающую все непустые действия $\alpha_{i_j} \neq \tau$.
Тогда $traces_\pi(P)$ – это множество трасс для всех возможных редукций процесса *P*. Определим $traces_\pi(P, D)$, где $D \subseteq F \cup A$, как множество трасс, в которых уделены метки не из множества *D* (т.е. включаем только $\alpha_{i_j} \in D$).

4.2. Общий метод трансляции в многопоточную программу

На входе общего метода трансляции А.1 имеется модель окружения в виде π-процессов и исходный код драйвера. В модели окружения есть два набора процессов верхнего уровня: первые – статические, существующие в единственном экземпляре и не использующие сигналы создания копии процесса, вторые – динамически порождаемые, создающиеся при получении сигнала копирования (например $!\,create_1(x_1).C_1(x_1)$).
У процесса верхнего уровня есть состояние, которое при трансляции моделируется структурой, например для процесса P_{module} будет сгенерирована структура *L_state*:

```
struct L_state {
  struct list list;
  enum L_states state;
  int p;
  int L4_param;
}
```

Состояния хранятся в списке *L_state_list* для того, чтобы иметь возможность порождать неограниченное количество копий процесса. Поле *state* – отражает текущий процесс нижнего уровня или дополнительное состояние *STOP*, означающее процесс 0. Например, для процесса P_{module} это *enum L_states {STOP, L1, L2, L3, L4, L5}*. Поле p – параметр динамических меток, генерируемый при создании процесса с помощью vp. Также структура включает в себя параметры процессов нижнего уровня. Например, *L4_param* – параметр процесса L4, которому передается возвращаемое значение функции init.
Порождение экземпляра процесса происходит либо в функции main, если процесс статический, либо при приеме сигнала создания копии процесса, если процесс динамический.

Функция, отвечающая за создание процесса, выделяет память под его состояние, инициализирует его функцией *L_init* и добавляет в список состояний *L_state_list*, например для процесса P_{module}:

```
void create_L(int p) {
   struct L_state *s = malloc(sizeof(struct L_state));
   L_init(p, s);
   list_add(&L_state_list, &s->list)
   pthread_create(L_thread, s);
}
```

Функция *L_init* инициализации устанавливает начальное состояние L1.

```
L_init(int p, struct L_state *s) {
  s->state = L1;
  s->p = p;
}
```

Далее в функции *create_L* создается поток процесса с помощью функции *pthread_create* библиотеки *pthreads*, выполняющий функцию *L_thread*. Функция *L_thread* в бесконечном цикле выполняет шаг процесса *L_step*, пока процесс не придет в состояние *STOP*. Так как потоки процессов выполняются параллельно, то необходимо захватить блокировку, чтобы шаг процесса выполнился атомарно.

```
void L_thread(struct L_state *s) {
  while(true) {
    lock();
    int should_stop = L_step(s);
    if(should_stop) {
      list_del(L_state_list,s);
      free(s);
      unlock();
```

```
      break;
    };
    unlock();
  }
}
```

На каждом шаге процесса происходит выбор одного из действий, выполняемых процессом: посылка сигнала, условный переход или -действие.

Посылка сигнала осуществляется с помощью вызова функции обработки сигнала.

Например, рассмотрим выдержки из функции *L_step* – шаг процесса P_{module}.

```
int L_step(struct S_state *s) {
  switch(nondet_int()) {
    case 0:
      if(s->state == STOP) return 1;
    break;
…
    case 3:
      if(s->state == L2) {
        int res = mstop();
        if(res == 0) {
          //Переход в состояние L5
          s->state = L5;
        }
      }
      break;
…
```

```
 }
 return 0;
}
```

В случае перехода в состояние STOP, при выполнении условия s->state==STOP, функция возвращает 1, что означает завершение процесса.

Для действий посылки сигналов сначала проверяется текущее состояние, в котором осуществляется посылка. Например, для посылки сигнала mstop:

$$L2 := \overline{mstop}.\langle L5\rangle\ldots$$

проверяется нахождение в состоянии L2 (s->state == L2). Если условие выполнено, то осуществляется вызов функции обработки приема сигнала *mstop*. Сигнал считается принятым, только если функция обработки вернула 0, иначе сигнал не принят ни одним из процессов и необходимо перевыполнить посылку. Таким образом, процесс отправитель ожидает пока процесс получатель перейдет в состояние, в котором он готов принять соответствующий сигнал.

В функциях обработки приема сигналов для всех процессов происходит попытка обработать входной сигнал. Для каждого типа процессов перебираются все имеющиеся экземпляры состояний процессов из списка и вызывается соответствующая функция обработки приема. Например, рассмотрим выдержки из функции приема mstop.

```
int mstop() {
 switch(VERIFIER_nondet_int()) {
 …
  case M:
   int k = list_size(M_state_list);
   if(k == 0) {return -1;}
   int i = nondet_int(k-1);
   struct M_state *s = list_get(M_state_list, i);
   int r = M_mstop(s);
   if(r == 0) return 0;
   else return -1;
```

```
    …
}
```

В функции *mstop* недетерминированно выбирается процесс получатель, для каждого из которых недетерминировано выбирается состояние одного из экземпляров в списке *M_state_list*. Далее вызывается функция обработки сигнала *M_mstop* для процесса $P_{trymoduleget}$.

Функции обработки сигнала передается состояние экземпляра процесса и параметры сигнала. Например, для процесса $P_{trymoduleget}$ генерируется следующая функция *M_stop*:

```
int M_mstop(struct M_state *s) {
  int res = -1;
  if(s->state == M && s->param == 0) {
     s->state = MD;
     res = 0;
  }
  if(s->state == MD) {
     s->state = MD;
     res = 0;
  }
  //установить res в 0 если сигнал обработан
  return res;
}
```

При получении сигнала *mstop* в состоянии *M(0)*, соответствующая проверка *s->state == M && s->param == 0*, осуществляется переход в состояние *MD*, в котором последующие сигналы захватов *tmg* возвращают $tmg^{ret}(false)$. В состоянии *MD* сигнал обрабатывается, но состояние не меняется.

Для связывания исходного кода драйвера с π-процессами, генерируется код для процессов-оберток. Рассмотрим процессы P_{fcall}, для процессов P_{init} и P_{exit} генерируется аналогичный код.

$P_{fcall} := !f(p_i, fptr_i, ctx_i, params_i).CALLED(p_i, fptr_i, ctx_i, params_i)$

$CALLED(p_i, fptr_i, ctx_i, params_i) :=$

$\tau.EXECUTING(p_i, fptr_i, ctx_i, params_i, result)$

Процесс *EXECUTING* имеет специальное значение, его выполнение определяется кодом драйвера, вызывается функция драйвера $fptr_i$.

```
int FCALL_step(struct M_state *s) {
  case 1:
    //lock should be held here
    if(s->state == CALLED) {
      s->state = EXECUTING;
      fptr = s->fptr; //fptr – вызываемая фукнция драйвера,
      ctx = s->ctx; //ctx – контекст вызова
      params = s->params; //params – параметры функции
    } else { break; }
    unlock();
    type y = *fptr(ctx, params);  //EXECUTING: вызов функции fptr
    lock();
    s->state = RET;
    s->RET_result = y;
  break;
  case 2:
    if(s->state == RET) {
      //Вызов функции действия с заданным параметром
      int res = f^ret(s->p,s->RET_result);
      if(res == 0) {
```

```
            //Переход в состояние STOP
            s->state = STOP;
        }}
        break;
    }
    return 0;
}
```

После того, как функция драйвера выполнилась, процесс переходит в состояние посылки возвращаемого значения *RET*, значение сохраняется в *RET_result*. В состоянии RET происходит посылка сигнала возврата, в случае успеха, процесс завершается.

В процессе выполнения функции драйвер может вызывать библиотечные функции сердцевины ядра, для которых генерируется модель, осуществляющая посылку сигналов соответствующим процессам. Например, для функции usb_register, генерируется следующий код посылки сигнала start(usbpn_driver, usbpn_driver->probe, usbpn_driver->disconnect) процесса P_{usb_driver}.

4.3. Модификация метода для трансляции в многопоточную программу

В последовательном случае при создании процесса, создается лишь экземпляр состояния и увеличивается счетчик активных процессов, создания потока с помощью pthread_create не происходит.

```
void create_L(int p, f_ptr init, f_ptr exit) {
    …
    //pthread_create(L_thread, s);
    thread_cnt++;
}
```

В функции *main* осуществляется недетерминированный выбор одного из типов процессов, добавляется бесконечный цикл *while(true)*.

```
void main() {
```

```
int p = globalId++;
create_L(p);//Создать основной поток
create_M(0);//Создать поток trymoduleget
while(true) {
  switch(nondet_int()) {
  case L_label:
    lock();
    int stop_loop = L_step_any();
    unlock();
    if(stop_loop) { goto break_loop;}
    break;
  }
  case …
 }
 break_loop:
}
```

Функции вида P_i*_step_any()* выбирают один из экземпляров процесса P_i и запускают для него *L_step*. В коде ожидания возврата из библиотечных функций наряду ожиданием возврата добавляется возможность совершить действие одному из других процессов, добавляется недетерминированный выбор экземпляров процессов аналогично бесконечному циклу в функции *main*.

4.4. Упрощения генерируемого кода для верификации

Для того чтобы успешно верифицировать генерируемый код с помощью современных инструментов высокоточной верификации, необходимо чтобы он не содержал неподдерживаемых конструкций. В первую очередь, современные верификаторы имеют ограничения при работе потенциально

неограниченными структурами данных в куче, такими как списки, деревья, и т.д. Поэтому необходимо минимизировать использование списков при генерации. На это нацелено первое упрощение, заключающееся в замене списков для хранения экземпляров состояний статических процессов на глобальные переменные. Например, для процесса P_{module}:

```
struct L_state L_state_0;
```

Для динамически порождаемых процессов ограничивается количество создаваемых экземпляров. Если количество экземпляров превышает максимальное, то новые экземпляры не порождаются.

Функции, работающие с экземплярами состояния процесса, размножаются по количеству копий, каждая копия работает со своей глобальной переменной, например вместо *L_step* генерируется *L_step_0*. В функциях P_i*_step_any* недетерминировано вызываются сгенерированные функции для каждого экземпляра.

Второе упрощение связано с тем, что инструменты верификации имеют ограниченную поддержку указателей на функции, поэтому в сгенерированном коде вызовы функций по указателю заменяются на явные вызовы. Это осуществляется генерацией для каждой функции драйвера своего процесса обертки с ее явным вызовом. Конечно, в местах вызова функций по указателю должно быть заранее известно, какая функция вызывается. В примере выше связывание указателей на функции с их значениями происходит при регистрации группы *usb_driver* и посылке сигнала *start(usbpn_driver, usbpn_driver->probe, usbpn_driver->disconnect)*. В этом месте создается экземпляр процесса P_{usb_driver}, который вместо посылки сигналов вызова функций по указателю *probe, disconnect* осуществляет посылку конкретным экземплярам *usbpn_probe, usbpn_disconnect* процесса P_{fcall}.

4.5. Теорема об эквивалентности

Далее мы покажем, что получающаяся в результате трансляции последовательная Си программа порождает те, и только те трассы взаимодействий, которые были в π-процессах. Так как исходный код драйвера в последовательном случае выполняется в одном потоке, то мы будем требовать, чтобы окружение в виде π-процессов одновременно осуществляло вызов только одной функции драйвера, т.е. после посылки сигнала f и до приема сигнала возврата f^{ret}, недопустимо посылать еще один сигнал *f*.

Определим понятие трассы выполнения для последовательной Си программы *CP*. Трасса Си программы определяется событиями вызова функции приема сигнала в сгенерированном коде. Данные функции генерируются в пункте 2.3 приложения А метода трансляции. Функции приема сигналов для каждой метки $a[p](y) \in A \cup F$ имеют вид:

```
int a(int p, type y)
```

Функции возвращают значение *0*, если сигнал успешно принят, и *-1* иначе. Событие $a[p](y)$ добавляется в трассу, если функция $a(p,y)$ завершается с кодом *0*. Множество всех трасс обозначим как $traces_C(CP)$. Определим $traces_C(CP, D)$, где $D \subseteq F \cup A$, как множество трасс, в которых удалены метки не из множества *D*.

Определим понятие трассовой эквивалентности для π-процесса P и сгенерированной Си программы *CP*.

Определение. π-процесс P трассово-эквивалентен сгенерированной Си программе CP на множестве событий D, обозначим как $P \approx_D CP$, если $traces_\pi(P, D) = traces_C(CP, D)$.

Теорема.

Пусть D – множество событий приема сигналов между драйвером и окружением

$$D = \{init, init^{ret}\} \cup \{exit, exit^{ret}\} \cup \{f, f^{ret}\} \cup \{g_1, g_1^{ret} \dots, g_l, g_l^{ret}\} \cup$$

$$\{set_{v_i}\} \cup \{get_{v_i}\}.$$

Пусть Sys_π – модель в π исчислении, состоящая из процессов окружения и процессов драйвера, т.е. $Sys_\pi = Env_\pi \mid Drv_\pi$.

Пусть Sys_C – Си код, состоящий из кода, сгенерированного для окружения Env_C (по методу А.2 [8]), и кода драйвера Drv_C.

Пусть $Drv_\pi \approx_D Drv_C$, т.е. код драйвера трассово-эквивалентен π модели.

$Тогда\ Sys_\pi \approx_D Sys_C$.

Таким образом, для любой трассы π процессов существует эквивалентная трасса в Си программе, и наоборот для любой трассы Си программы существует эквивалентная трасса π-процессов.

Доказательство теоремы приведено в разделе А.3 приложения диссертации [8].

Заключение

Предложенный в работе подход к описанию окружения драйвера позволяет с одной стороны, иметь компактные описания в виде π-процессов, которые позволяют абстрагироваться от многих деталей окружения, допуская разнообразные сценарии взаимодействия. С другой стороны, выразительная мощность π-процессов позволяет описывать необходимые ограничения на взаимодействия окружения с драйвером.

Метод трансляции π-процессов обеспечивает необходимые требования инструментов высокоточного статического анализа, так как в результате

трансляции получается последовательная Си программа, что позволяет осуществлять ее верификацию данными инструментами.

Описанные в работе подходы к моделированию окружения драйверов используются в проекте верификации драйверов ОС Linux [9].

Литература

[1]. N. Palix, G. Thomas, S. Saha, C. Calves, J. Lawall, and Gilles Muller. Faults in Linux: Ten years later. Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems (ASPLOS '11), USA, 2011.

[2]. D. Beyer, T. Henzinger, R. Jhala, R. Majumdar. The Software Model Checker Blast: Applications to Software Engineering. International Journal on Software Tools for Technology Transfer (STTT), vol. 5, p. 505-525, 2007.

[3]. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7 //Proceedings of TACAS. 2012. Vol. 7214. P. 525–527.

[4]. D. Beyer, M.E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg, 2011.

[5]. T. Ball, E. Bounimova, R. Kumar, V. Levin. SLAM2: Static Driver Verification with Under 4% False Alarms. FMCAD, 2010.

[6]. J. Corbet, G. Kroah-Hartman, A. McPherson. Linux kernel development. How Fast it is Going, Who is Doing It, What They are Doing, and Who is Sponsoring It. http://go.linuxfoundation.org/who-writes-linux-2012, 2012.

[7]. Milner R. The Polyadic π-Calculus: a Tutorial. LFCS, Department of Computer Science, University of Edinburgh, 1991. P. 49.

[8]. Мутилин В.С. Верификация драйверов операционной системы Linux при помощи предикатных абстракций. Диссертация на соискание ученой степени к.ф.-м.н., 2012.

[9]. Мандрыкин М. У., Мутилин В. С., Новиков Е. М. и др. Использование драйверов устройств операционной системы Linux для сравнения инструментов статической верификации //Программирование. 2012. Т. 5. С. 54–71.

Environment Modeling of Linux Operating System Device Drivers

Zakharov I.S., Mutilin V.S., Novikov E.M., Khoroshilov A.V.
{ilja.zakharov, mutilin, novikov, khoroshilov}@ispras.ru
ISP RAS, Moscow, Russia

Abstract. To establish verification of Linux operating system device drivers it is necessary to take into account particularity of communication between drivers and a kernel core as far as it plays the main role in drivers behavior. At the same time, verification of a driver together with kernel core source code is not feasible due to complexity and size of the resulting code. That is why we suggest to replace the real environment with its model. Such model of the environment should be correct and complete. Correctness means that it should contain only interaction scenarios which can happen in the real environment. Completeness means that all feasible in the real environment scenarios should present in the corresponding model. Moreover, the environment model should be able to be checked by verification tools. Hence, it should be translated in equivalent representation supported by these tools.

The paper presents a new method for modeling driver environment based on R. Milner calculus and a method of model translation into a program in the C programming language. Being linked with driver source code this program describes the same scenarios of driver behavior as the real driver environment in the operating system.

Keywords: operating system; driver; environment; verification.

References

[1]. Beyer D. Second Competition on Software Verification. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7795, pp. 594-609, 2013. doi: 10.1007/978-3-642-36742-7_43

[2]. Beyer D., Henzinger T., Jhala R., Majumdar R. The Software Model Checker Blast: Applications to Software Engineering. International Journal on Software Tools for Technology Transfer (STTT), vol. 5, pp. 505-525, 2007. doi: 10.1007/s10009-007-0044-z

[3]. Shved P., Mandrykin M., Mutilin V. Predicate Analysis with Blast 2.7. In Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS), LNCS, vol. 7214, pp. 525–527, 2012. doi: 10.1007/978-3-642-28756-5_39

[4]. Beyer D., Keremoglu M.E. CPAchecker: A Tool for Configurable Software Verification. In Proc. Computer Aided Verification (CAV), LNCS, vol. 6806, pp. 184–190, 2011. 10.1007/978-3-642-22110-1_16

[5]. Milner R. The Polyadic π-Calculus: a Tutorial. In Proc. Logic and Algebra of Specification, NATO ASI Series, vol. 94, pp 203-246, 1993. doi: 10.1007/978-3-642-58041-3_6

[6]. Mutilin V.S. Verifikatsiya drajverov operatsionnoj sistemy Linux pri pomoshhi predikatnykh abstraktsij [Linux drivers verification with help of predicate abstractions]. Dissertatsiya na soiskanie uchenoj stepeni k.f.-m.n. [PhD thesis], 2012 (in Russian).

[7]. Mandrykin M.U., Mutilin V.S., Novikov E.M., Khoroshilov A.V., Shved P.E. Using linux device drivers for static verification tools benchmarking. Programming and Computer Software, vol. 35, issue 5, pp. 245-256, 2012. doi: 10.1134/S0361768812050039

[8]. Mutilin V.S., Novikov E.M., Strakh A.V., Khoroshilov A.V., Shved P.E. Arkhitektura Linux Driver Verification [Linux Driver Verification Architecture]. Trudy ISP RAN [The Proceedings of ISP RAS], vol. 20, pp. 163-187, 2011 (in Russian).

[9]. Khoroshilov A., Mutilin V., Novikov E., Shved P., Strakh A. Towards an Open Framework for C Verification Tools Benchmarking. In Proc. Perspectives of Systems Informatics (PSI), LNCS, vol 7162, pp. 82-91, 2012. doi: 10.1007/978-3-642-29709-0_17

Оценка эффективности минимизации ограничений запросов к СУБД

Кузнецов С. Д., kuzloc@ispras.ru, Мендкович Н. А., mend@f-group.ru

Аннотация: Эта статья описывает усовершенствованные алгоритмы лексической оптимизации запросов. Алгоритмы обнаруживают и удаляют избыточные условия из ограничения запроса, чтобы упростить его. Статья также представляет результаты применения этих оптимизационных техник и их влияние на скорость обработки запроса.

Ключевые слова: оптимизация запросов, лексическая оптимизация, выполнение запросов.

1. Введение

Вследствие эволюции вычислительной техники в течение последних десятилетий увеличивается роль таких приложений, как системы управления базами данных (СУБД). Важную роль в функционировании современных СУБД играют высокоуровневые языки запросов. Они позволяют формулировать и эффективно решать различные задачи, связанные с поиском и обработкой информации, хранящейся в базах данных.

С ростом объема хранимых данных и усложнением управленческих процессов, которые требуют выполнения сложных запросов, затрагивающих большие объемы хранимых данных, нетривиальной становится задача организации БД и повышения эффективности обработки запросов. Возникают новые проблемы функционирования СУБД, касающиеся недостаточной быстроты выполнения мощных операций, необходимых для обеспечения приемлемого уровня производительности системы. В особенности это касается повышения скорости обработки запросов на выборку данных. В рамках разрешения последней проблемы в области исследования баз данных выделилась область называемая *оптимизацией запросов*, которой посвящено значительное число работ [1-5].

Для дальнейшего изложения необходимо условиться о значении используемых терминов. Будем называть *запросом* любое языковое выражение, которое описывает совокупность данных в базе данных (БД), подлежащих выборке или обновлению. В запросе мы выделяем *ограничение* – часть запроса, содержащую все множество условий, описывающих выборку,

выдачей которой должно завершаться его исполнение. *Условие* – часть ограничения запроса, описываемая с помощью множества атрибутов запроса и констант и содержащая не более чем одну операцию сравнения. Каждое условие рассматривается как лексическая конструкция, поэтому в его рамках выделяется *корень* – множество входящих в его состав атрибутов. Помимо корня в состав условия входят операции преобразования и сравнения значений, а также константы.

Исследователи предлагают значительное число различных алгоритмов, в том числе, основанных на генерации и поиске оптимального плана доступа, выборе наиболее экономных потенциальных и низкоуровневых процедур и модификации самого текста запроса с целью приведения к наиболее оптимальной форме.

Последний вид оптимизации (модификация) подразделяется на два вида оптимизирующих преобразований: семантические, основанные на анализе содержимого БД и существующих в ней ограничений целостности, и лексические, оперирующие исключительно текстом самого запроса. Последний класс операций может быть признан наиболее широко применимым особенно в случае распределенной обработки запроса, так как оптимизация осуществляется без обращения данным БД и не связана с издержками при передаче данных.

Существует три вида лексической оптимизации запросов:

-перезапись (в частности, перемещение элементов по дереву запроса[6, Pp. 354-366], удаление вложенных подзапросов и проч. [7, Pp. 147-152; 8, Pp. 91-102]);

-украшение (введение в текст запроса дополнительных элементов ускоряющих обработку, т.н. «магические множества»[9]);

-сокращение (исключение избыточности в условии запроса).

Последнему классу оптимизирующих алгоритмов в современных работах уделяется недостаточно внимания[10, Cc. 204-207], несмотря на то, что не исчерпаны возможности повышения эффективности оптимизации и расширения множества поддающихся обнаружению и устранению видов избыточности.

Судя по опубликованным данным о принципах работы оптимизаторов широко распространенных СУБД, с целью удаления избыточных условий в них реализуются только алгоритмы поиска общих подвыражений в конъюнктах условий. В частности, это касается MySQL[11-12], PostgreSQL[13] и Oracle[14, P. 9].

Отказ от иных способов оптимизации, например, обработки дизъюнктов ограничений, можно рассмотреть на примере работы оптимизатора из описания MySQL 5.5[12], где при описании приемов лексической оптимизации приводится пример преобразования:

((a AND b) AND c OR (((a AND b) AND (c AND d)))) = >

(a AND b AND c) OR (a AND b AND c AND d).

Однако очевидно, что данное преобразование не завершено и итоговое кратчайшее представление данной функции

a AND b AND c,

так как X OR (X AND a) = > X[1].

Подобная логическая незавершенность преобразования ограничения демонстрирует наличие значимых нерешенных проблем в области идентификации избыточных условий в запросах.

В целях обнаружения, идентификации и устранения избыточности нами разработана процедура, включающая 3 основных операции:

- оптимизация на основе поглощения дублирующих друг друга условий в составе запроса;
- оптимизация набора условий с использованием алгоритмов минимизации булевой алгебры;
- оптимизация ограничения с помощью алгоритмов линейной алгебры.

Описание представленных ниже алгоритмов ранее публиковалось авторами[15-17]. В данной главе их описание приводится комплексно с представлением примеров реализации.

2. Описание алгоритмов

2.1. Алгоритм поглощения

Задача поглощения элементарных избыточных условий на основе поиска общих подвыражений в ограничении поставлена еще в наиболее ранних работах, посвященных оптимизации запросов[18, P. 247].

Эти преобразования являются одной из «ключевых технологий трансформации» запросов в оптимизаторах ряд СУБД, включая Oracle[14, P. 21; 19, P. 1368] и MySQL[11]. Однако они сводятся к достаточно тривиальному поиску избыточных условий, удаление одного из которых не изменяет смысла ограничения, но делает его более лаконичным.

Однако в существующих публикациях отсутствует подробное описание работы этих алгоритмов, поэтому неясно, сколь широкий круг избыточностей они могут устранять. Приводимые в упомянутых источниках примеры позволяют предположить, что указанные СУБД могут находить различные

[1]Данная ошибка присутствует во всех изданиях СУБД MySQL, включая версию для MySQL 5.5, изданную в 2013 году.

избыточные выражения среди условий, являющихся строгими тождествами (*arith-expr1 comp-op arith-expr2*). Наконец, в упомянутых источниках отсутствует подробное описание критериев выявления избыточности и процедуры их устранения, что затрудняет оценку эффективности этих алгоритмов.

В этом разделе описывается алгоритм, позволяющий находить и сокращать широкий набор условий, являющихся избыточными, но не совпадающих текстуально даже после стандартизации выражения. Для этого требуется перейти от уровня формального синтаксиса выражений до их семантики, сравнивая их части между собой. Избыточность может выражаться в том, что один из конъюнктов ограничения, записанного в ДНФ[2], избыточен по отношению к другому, так как все значения, описанные первым конъюнктом, входят в число значений, описанных вторым конъюнктом.

В рамках поиска таких избыточностей производится сравнение конъюнктов, содержащих условия с одинаковой левой частью (*arith-expr1*), и приводится их логическое сопоставление с целью выявления случаев, когда «однокоренные» условия могут поглощать друг друга из-за отсутствия условий, не являющихся общими или не поглощающими друг друга.

Например, запрос

```
SELECT ID_пользователя
FROM EMP
WHERE (ID>0 AND Salary>500 AND Salary+Bonus<1000) OR (Salary>400);
```

является неоптимальным, и может быть представлен в более лаконичной и оптимальной форме:

```
SELECT ID_пользователя
FROM Пользователи
WHERE Salary>400;
```

Для данного вида оптимизации сформулирована система правил поглощения одними условиями других в рамках описанной модели. Эти правила представлены в таблице 2.1, где первая строка соответствует поглощающему условию, первый столбец – поглощаемому, а в каждом из полей представлено необходимое и достаточное условие для указанного поглощения. Прочерк означает невозможность поглощения в данном случае.

[2]В рамках нашего алгоритма все условия запроса приводятся к дизъюнктивной нормальной форме, поэтому, обсуждая структуру ограничения, мы оперируем иерархией – условие, конъюнкт, ограничение в виде дизъюнкта конъюнктов.

Таблица 2.1. Правила поглощения условий.

	y == const2	y > const2	y < const2
x == const1	const2 == const1	const1 > const2	const2 > const1
x > const1	-	const1 > const2	-
x < const1	-	-	const2 < const1

Более подробное описание алгоритма поглощения представлено в другой нашей работе[16, Сс. 26-28].

2.2 Алгоритм Квайна

Ряд приемов по преобразованию логических выражений к более краткой форме разработан в рамках булевой алгебры. Они основаны на идее «склейки» пар конъюнктов, содержащих выражение и его отрицание при общих прочих одинаковых членах. Применимые для решения этой задачи способы минимизации логических функций были разработаны еще в начале 1950-х гг. [20-21], (т.е. еще до начала специальных исследований в области оптимизации запросов, которые относится к началу 1970-х гг.), позже они были адоптированы для программной реализации У. Квайном и И. Маккласки[22-23].

Возможность использования алгоритма Квайна-Маккласки при оптимизации запросов допускалась в ряде более поздних работ[24, P. 234; 25-26], однако описания примеров его реализации в данных целях нам обнаружить не удалось.

Для выполнения алгоритма Квайна требуется выделить в ограничении группу конъюнктов, содержащих одинаковое количество условий (например, n). Затем проводится попарное сравнение конъюнктов и «склейка» тех пар, которые имеют вид (A>B) AND X и NOT(A>B) NOT X, где X – некий конъюнкт условий размерностью n-1. Конъюнкты X именуются *первичными импликантами* и записываются отдельно. Затем для всех выбранных X повторяется процедура попарного анализа и «склейки» с целью получения конъюнктов размерностью n-2, и так до тех пор, пока удается обнаруживать пары конъюнктов вида (A>B) AND X и NOT(A>B) NOT X.

Мы предлагаем следующее программное представление данного алгоритма. Его целью является определение минимального множества импликант, соответствующего максимальному множеству конъюнктов, с целью их замены в конечном представлении. Для этого образуется таблица где строкам соответствуют первичные импликанты всех размерностей, а столбцам – конъюнкты исходного представления ограничения. Элемент таблицы имеет значение 0, если первичная импликанта данной строки не входит в конъюнкт, соответствующий столбцу, и 1 – если входит.

Для определения искомого набора импликант производятся пять этапов обработки этой таблицы:

1. Из дальнейшего анализа исключаются столбцы, содержащие только нули. Они в любом случае входят в конечное представление, поскольку не покрываются найденными импликантми.
2. Производится поиск *существенных импликант*, т.е. строк, одно из единичных значений которых является единственным в столбце. Обнаруженные существенные импликанты и соответствующие им конъюнкты исключаются из дальнейшего анализа
3. В видоизмененной таблице выделяются одинаковые столбцы, содержащие единицы в одних и тех же строках. Все столбцы-«дубликаты», кроме одного, исключаются из таблицы.
4. Исключаются все строки, не содержащие единичных значений (их образование возможно после завершения предыдущего этапа).
5. Из числа оставшихся строк выбирается набор импликант, включающий единицы во всех столбцах. Если имеется несколько вариантов, то выбирается тот, который содержит минимальное число условий.
6. Формируется итоговое представление ограничения, состоящие из конъюнктов, выбранных на этапе 1, и импликант, выбранных на этапах 2 и 5.

В рамках решения задачи оптимизации запросов данный алгоритм использует иное основание для «склейки» конъюнктов, нежели поиск пар «утверждение-отрицание». Учитывая, что в рамках анализа запросов происходят операции с непосредственным содержанием условий, в качестве критерия «склейки» мы используем признак тождественной истинности логической суммы двух утверждений.

Рассмотрим работу алгоритма на следующем примере. Пусть требуется отобрать авторов, имеющих различные соотношениям чисел опубликованных статей, книг и цитирований их работ другими авторами. Ограничение данного запроса будет иметь следующий вид:

...WHERE Articles>25 AND Books>2 AND Citations=32

OR NOT(Articles>25) AND Books>2 AND Citations=32

OR NOT(Articles>25) AND Books>2 AND NOT(Citations=32)

OR Articles>25 AND NOT(Books>2) AND NOT(Citations=32)

В результате применения описанного выше алгоритма выражение примет следующую более краткую форму:

Books>2 AND Citations=32

OR NOT(Articles>25) AND Books>2

OR Articles>25 AND NOT(Books>2) AND NOT(Citations=32)

2.3 Алгоритм оптимизации линейных неравенств

Также нами разработан алгоритм минимизации ограничений запроса содержащих условия с неодинаковой левой частью (*arith-expr1*), но имеющих общие атрибуты. Данный алгоритм аналогичен подходам, представленным в линейной алгебре, однако сам является абсолютно новым.

Большинство существующих в данной области работ посвящено методам минимизации систем линейных равенств[28]. К их числу относятся ставший классический алгоритм, известный как «метод Гаусса» или «метод Гаусса-Зейделя» (подробное описание представлено в [29, сс. 159-162]). Методы, разработанные для решения задач оптимизации систем линейных неравенств, кроме описанного ниже, – авторам неизвестны. Однако возникновение подобных задач применительно к оптимизации запросов представляется возможным.

Пусть имеется ограничение запроса, представленное в виде конъюнкции условий, которое направлено на то, чтобы отобрать значения, соответствующие некоему набору неравенств. Примером этого может служить следующее ограничение:

...WHERE (Salary+Bonus>10000

AND Salary-Payment>3000

AND 2*Payment>8000)

OR Payment-Bonus=>100,

оптимальной формой представления которого является выражение:

Payment-Bonus=>100

Алгоритм оптимизации подобного рода ограничений основан на попарном анализе всех условий конъюнкта, имеющих общие атрибуты. На основе этой процедуры создаются новые временные условия-«следствия». Кроме того, в текст включаются внешние условия, т.е. неравенства обратные одиночным условиям, объединенным с анализируемым конъюнктом операцией логического сложения (в данном примере это условие Payment-Bonus=>100).

Формулирование следствий осуществляется на основе системы формальных правил, часть из которых представлена в таблице 3. В ней первый столбец соответствует парному неравенству, первая строка – рабочему, а все прочие клетки общему виду следствия для конкретного случая. Прочерк означает, что на основе стандартного алгоритма в данном случае невозможно вывести следствие значимое для оптимизации.

Таблица 2.2. Общий вид определения следствий для пар неравенств, где левая часть парного полностью входит в рабочее неравенство.

	a+b= const1	(a-b=const1)	(a+b>const1)	(a+b<const1)	(a-b>const1)	(a-b<const1)
(b> const2)	a< const1- const2	a>const2+ const1	-	a<const1- const2	a>const1+ const2	-
(b< const2)	a> const1- const2	a<const2+ const1	a>const1- const2	-	-	a<const1+ const2
(a> const2)	b< const1- const2	b>const2- const1	-	b<const1- const2	-	b>const2- const1
(a< const2)	b> const1- const2	b<const2- const1	b>const1- const2	-	b<const2- const1	-

При равенстве a=const переменная a заменяется на константу из правой части во всех условиях, поэтому не учитывается в представленной выше таблице.

После серии преобразований анализируемый конъюнкт расширяется за счет включения условий следствий и внешних условий. После этого проводится формальной поиск логических противоречий и избыточных выражений, являющихся подмножеством следствий из других пар условий или внешних выражений, включенных в конъюнкт.

В итоге операции все следствия и внешние условия удаляются из конъюнкта вместе с выражениями, признанными избыточными. В случае обнаружения логических противоречий, делающих конъюнкт невыполнимым, из обработки исключается сам конъюнкт.

Более подробное описание реализации данного алгоритма представлено в предыдущей нашей работе[17, сс. 144-154].

3. Оценка эффективности алгоритма

С точки зрения практической эффективности преобразование запроса или плана его выполнения может привести к изменению стоимости его обработки в большую или меньшую сторону.

Возможны следующие определения эффективности алгоритма:

-оптимизирующий, т.е. ведущий к сокращению затрат на обработку запроса путем сокращения использования машинных ресурсов, превышающего стоимость выполнения алгоритма оптимизации;

-равноценный, описывающий случаи, когда экономия, связанная с оптимизацией запроса, и затраты на реализацию алгоритма – равны;

-убыточный, т.е. случай, когда расходы, связанные с выполнением оптимизирующего алгоритма, превышают достигнутую экономию, или же само осуществленное преобразование исходного запроса приводит к его «удорожанию» в сравнении с изначальной стоимостью.

С целью определения эффективности разработанных алгоритмов проведена серия экспериментальных испытаний их влияния на скорость обработки запросов, итоги которой приведены в настоящем разделе. Ее текст включает описание существующих критериев эффективности и обоснование выбора одного из них. Кроме того, дается описание тестовой среды, условий эксперимента и анализа его результатов.

3.1 Тестирование алгоритмов и оценки их эффективности

Для оценки эффективности разработанных алгоритмов необходимо проведение анализа стоимости, т.е. затрат системных ресурсов на их реализацию. Эффективность оптимизации определяется затратами на выполнение оптимизированного и неоптимизированного вариантов запроса с учетом затрат на осуществление самого процесса оптимизации.

В результате возникает задача определения ресурсов, использование которых должно быть минимизировано при выполнении запросов, и их затрат на реализацию плана доступа и алгоритма оптимизации. Традиционно в качестве меры для оценки эффективности алгоритмов используются следующие критерии [1, Pp. 113-114]:

1. затраты на текущее хранение данных (storage cost), связанные с использованием памяти;
2. стоимость передачи данных, т.е. обмен данными между машинным комплексом, в рамках которого осуществляется хранение, и местом вычисления (communication cost), а также время обмена данными между вторичной и основной памятью (secondary storage access cost или I/O cost);
3. затраты, характеризующие использование центрального процессора, в связи с вычислением запроса (computation cost).

Учитывая динамику развития вычислительных средств значительно сокращается дефицит ресурсов оперативной памяти, поэтому ключевыми становятся показатели, характеризующие временные затраты на выполнение алгоритма.

Была предложена следующая функция вычисления стоимости обработки запроса, где в качестве единицы измерения используется время[30, P. 233]:

$$T_{total}=T_{CPU}+T_{I/O}+T_{MSG}+T_{TR}, \qquad (3.1),$$

где T_{total} – общие временные затраты, T_{CPU} – процессорное время, $T_{I/O}$ – время обмена данными между вторичной и основной памятью, T_{TR} – время коммуникации, T_{MSG} – время, затраченное на инициирование запросов и сообщений, а также обработку данных, полученных от внешних ресурсов. В случае, когда речь идет о локальной базе данных, последние два компонента могут быть признанными равными 0.

Со временем роль каждой из составляющих стоимости обработки запросов была пересмотрена. Прежде всего, в связи с ростом объемов памяти вычислительных систем стоимость хранения данных перестала быть «узким местом» [1, P. 144] и играть значимую роль в оценке эффективности алгоритмов оптимизации.

Стоимость передачи данных является все еще важным фактором в функционировании информационных систем, так как все большую популярность приобретают распределенные базы данных, а также распределенные вычислительные системы, что значительно повышает роль издержек, связанных с обменом данными в процессе реализации запроса. В некоторых работах данный вид затрат рассматриваются как единственный критерий эффективности алгоритма (например, [31, P. 97]).

Однако в случае предложенных нами алгоритмов ситуация является иной. Все процедуры, связанные с оптимизирующими преобразованиями запроса, осуществляются локально и не используют какие-либо данные, хранящиеся в базе данных, к которой будет адресован запрос, или характеризующие ее состояние. Предложенные алгоритмы оптимизации могут быть названы «автономными». Естественно, они в силу своих свойств не могут привести к увеличению затрат, связанных с обменом данными, но могут привести к их сокращению в результате устранения из запроса избыточных условий, обработка которых потребовала бы увеличения коммуникационных издержек.

На основе выше изложенных аргументов в качестве критерия оценки эффективности работа алгоритмов оптимизации нами выбраны вычислительные затраты. Следующим шагом является выбор конкретных характеристик, отражающих работу ЦП по обработке запроса и реализации алгоритма оптимизации.

В современных исследованиях большой популярностью пользуется подход, ориентированный на учет числа машинных операций, с помощью которого характеризуются затраты процессорного времени. В современных исследованиях существуют два основных подхода к оценке стоимости с их помощь [32, P. 177]:

-унифицированная ценовая модель (uniform cost model) в качестве единицы измерения рассматривающая одну машинную операцию;

-и логарифмическая ценовая модель (logarithmic cost model), где стоимость каждой машинной операции корректируется с учетом объема вовлеченных данных.

Первый подход пользуется широкой популярностью в виду простоты и иногда считается фактически безальтернативным [33, Pp. 26-27]. Однако доказано, что он может приводить к значительному искажению оценки стоимости процесса из-за различий в объемах данных, необходимых при реализации алгоритма[32, Pp. 177-179].

В рамках решения данной проблемы в качестве критерия стоимости работы алгоритма в нашем исследовании используется **время выполнения задачи процессором**. Использование времени в качестве средства измерения стоимости в данном случае представляется оптимальным и с точки зрения решения проблем «унифицированной» модели, и как универсальный критерий, позволяющий охарактеризовать все виды затрат, связанных с обработкой запросов [30, Pp. 233-234].

3.2 Описание тестовой среды

В рамках экспериментальной проверки была проведена серия испытаний в специально созданной тестовой среде, включающей в себя СУБД Oracle Database 10g Express Edition, а также программный комплекс «Оптимизатор», оптимизирующий условия запросов к реляционным базам данных.

Указанные программные продукты были установлены на персональном компьютере, оснащенном процессором Intel (R) Core i5-2430M с частотой 2,4 Ггц, установленной памятью 4 Гбайт, 64 разрядной операционной системой Windows 7.

Выбор Oracle Database 10g определялся широким распространением данного приложения, встроенными приложениями измерения затрат системного времени, доступ пользователя к генерируемым планам доступа, что облегчало проведение экспериментов. Погрешность измерения времени средствами СУБД составляет 0,01 секунды.

Программный комплекс включает в себя приложения, производящие минимизацию числа условий запроса с помощью алгоритмов, описанных выше: сокращение запроса с помощью поглощения избыточных условий, оптимизация по Квайну-Маккласки, а также оптимизация запроса как системы неравенств. «Оптимизатор» включал в себя приложения, позволяющие оптимизировать ограничение запроса как по одному из перечисленных выше критериев, так и с помощью всех описанных алгоритмов. В итоге программа генерирует оптимальный вид ограничения запроса.

При отладке «Оптимизатора» была проведена серия тестов, в рамках которых приложение преобразовало серию произвольных выражений. Результаты этих испытаний мы приводим ниже в табл. 3.1. Время, затраченное системой на оптимизацию выражения, указывается в микросекундах (μs). Кроме того, в таблице приводятся данные о числе элементарных условий в изначальной форме оптимизируемого выражения.

Таблица 3.1. Результаты тестирования программы «Оптимизатор»

Тест	Время оптимизации (μs)	Число условий
1	122,913	4
2	91,0689	9
3	33,7007	24
4	52,5365	4
5	93,5856	9
6	81,6848	7
7	81,3892	7
8	52,4639	4
9	48,4827	4

На основе представленных данных можно заключить, что программа-оптимизатор совершает обработку при сравнительно небольших затратах времени.

Причем указанные затраты демонстрируют слабую зависимость от числа условий в оптимизируемом выражении. Линейный коэффициент корреляции, рассчитанный на основе данных табл. 3.1, равен 0,37, что указывает на отсутствие явно выраженной статистической зависимости. Различия во временных затратах на преобразование запросов обусловлены функционирование фоновых процессов в период работы программы. Однако создаваемая ими погрешность не превышает 100 μs, что не оказывает влияния на итоги тестов, описанных ниже (раздел 3.3). Кроме того, данная погрешность равна неустранимой погрешности оценки времени обработки запроса средствами СУБД Oracle Database 10g.

Перед проведением тестирования в рамках СУБД была создана группа таблиц данных, имеющих идентичный набор атрибутов и объем данных. Каждая из созданных таблиц включала атрибуты, представленные в таблице 3.2.

Таблица 3.2. Структура таблицы, используемой в тестовой среде.

Атрибут	Тип данных
Name	Char(100)
Date	Date
N1	Number
N2	Number
N3	Number
N4	Number

В каждой таблице размещено 100 записей, содержащих различные значения перечисленных атрибутов. С целью оценки производительности системы с учетом оптимизации запросов была проведена серия тестов, включающих реализацию запросов на выборку данных, ограничение которых содержало бы о 2 до 4 атрибутов таблицы.

Осуществление запросов и учет времени их реализации производится средствами самой СУБД.

3.3 Результаты тестирования

В данном разделе мы описываем проведение тестирования и его результаты. Оно осуществляется с помощью реализации ряда запросов к описанной выше базе данных в среде Oracle Database 10g.

Все запросы были направлены на выборку равного числа атрибутов из таблиц одинаковой размерности и отличались исключительно составом ограничения. Каждый из них содержал те или иные виды избыточности. В запросах не используются вложенные подзапросы. Ограничение состояло из множества условий, представленных в дизъюнктивной нормальной форме.

Порядок эксперимента имел следующий вид:

- реализация запроса в первичной форме (время выполнения T_1);
- выполнение программы оптимизации запроса (T_{opt});
- реализация оптимизированного запроса (T_2).

После проведения перечисленных испытаний проводилось сравнение системного времени, затраченного на реализацию каждого из этапов (в терминах формулы 3.1 - T_{total}).

Эффективность алгоритма определялась соотношением

$$I_{eff} = (T_2+T_{opt})/T_1, \qquad (3.2)$$

где I_{eff} – коэффициент эффективности оптимизации.

Кроме того, целью эксперимента является оценка взаимосвязи числа условий в составе запроса со скоростью его обработки, для чего также проводится учет двух других показателей:

-число условий в составе запроса до оптимизации (A_1);

-число условий в составе запроса после оптимизации (A_2).

Результаты тестов представлены в таблице 3.4.

Таблица 3.3. Результаты тестов (время указывается в секундах).

	T_1	T_{opt}	T_2	I_{eff}	A_1	A_2
1	0,3	0,00008	0,1	0,3336	3	1
2	0,5	0,00011	0,1	0,2002	6	2
3	0,8	0,00008	0,15	0,1876	5	2
4	0,4	0,00010	0,2	0,5003	7	4
5	0,8	0,00008	0,2	0,2501	7	3
6	0,14	0,00007	0,03	0,2148	5	3
7	0,06	0,00008	0,01	0,1680	4	3
8	0,02	0,00009	0,009*	0,4545	9	6
9	0,23	0,00008	0,02	0,0873	3	1
10	0,15	0,00008	0,08	0,5339	3	1

*Менее 0,01 секунды, приводится максимально возможное значение из-за погрешности измерения.

Результаты тестирования показывают, что время обработки оптимизированного запроса значительно ниже, чем его обработка без проведения лексической оптимизации. Среднее ускорение обработки запроса времени в ходе проведенных тестов составляет 4,5 раза. (См. рисунок 1).

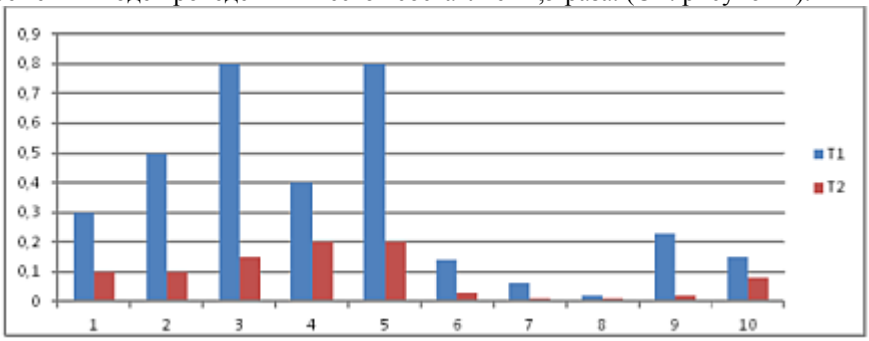


Рисунок 1. Время выполнения запроса до (T_1) и после оптимизации (T_2). (Время в секундах).

Коэффициент эффективности оптимизации демонстрирует возможность сокращения стомости обработки запроса на 47%-84%. Причем временные затраты на собственно оптимизацию запроса пренебрежимо малы в сравнении с затратами на его обработку. Проведенные тесты показывают, что они в 1000-10000 раз меньше, чем временные затраты системы на выполнение запроса.

Анализ причин роста скорости обработки запросов предполагает зависимость временных затрат от числа условий в составе ограничения запроса. Сравнение отношений показателей A_1 и A_2 с I_{eff} не позволяет установить однозначую статистическую зависимость, что, в частности, может быть обусловлено качественными отличиями между условиями и временем необходимым для их обработки.

Кроме того, статистическая зависимость может искажаться ошибкой измерения. При исключении из выборки тестов время обработки которых меньше 0,03 секунд (напомним, что точности измерения времени СУБД до 0,01 секунды), линейный коэффициент корреляции составляет 0,42, что демонстрирует прямую зависимость между этими двумя показателями.

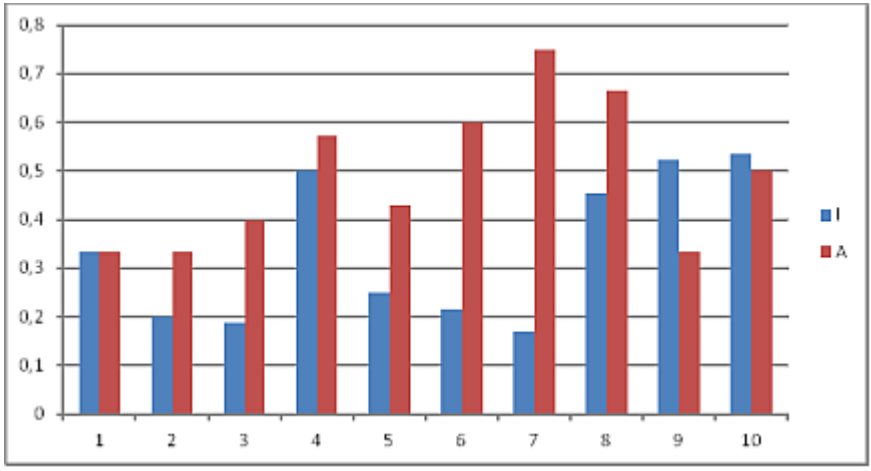


Рисунок 2. Коэффициент эффективности оптимизации запроса (I) и соотношение числа условий после и до проведения оптимизации ограничения (A).

Таким образом, проведенные тесты позволяют констатировать, что лексическая оптимизация запросов с помощью описанных выше алгоритмов позволяет сократить затраты на обработку запроса за счет сокращения избыточных условий.

При этом временные затраты на реализацию оптимизационных алгоритмов пренебрежимо малы в сравнении с временем обработки запроса СУБД. Это позволяет утверждать, что описанные в нашей работе средства лексической оптимизации позволяют повысить эффективность существующие систем оптимизации запросов.

Заключение

В данной работе были описаны алгоритмы лексической оптимизации запросов, позволяющие устранить виды избыточности, неопределимые существующими программами оптимизаторами.

Приводится подробное описание тестирования данных алгоритмов оптимизации и их влияния на стоимость обработки запроса. Анализируются существующие критерии оценки эффективности работа алгоритмов, осуществляется выбор критерия для данного исследования. Также описывается тестовая среда, включающая в себя СУБД Oracle Database 10g. В тестировании были учтены погрешности измерения и обработки, связанные с особенностями тестовой среды.

Проведенные тесты показывают, что временные затраты на реализацию разработанных алгоритмов лексической оптимизации запросов не превышают 0,0002 секунды, что в 1000 раз меньше времени выполнения рассмотренных запросов к базе данных. Это указывает на то, что выполнение указанных оптимизационных алгоритмов даже в случаях, когда оно не ведет к оптимизации запроса, не может вызвать значимое замедление его обработки.

Список литературы

[1]. Jarke M., Koch J. Query Optimization in Database Systems // ACM Computing Surveys (CSUR), 1984. March, Volume 16, Issue 2. Pp. 111-152.

[2]. Mannino M. V., Chu P., Sager T. Statistical profile estimation in database systems // ACM Computing Surveys, Volume 20, Issue 3. September 1988.

[3]. Ionnidis Y. E. Query Optimization // The Computer Science and Engineering Handbook. Boca Raton, CRC Press, 1996.

[4]. Chaudhari S. An Overview of Query Optimization in Relational Systems // Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, 1998.

[5]. Кузнецов С. Д. Методы оптимизации выполнения запросов в реляционных СУБД. [http://www.citforum.ru/database/articles/art_26.shtml]. [Обращение 20 ноября 2012].

[6]. Chaudhuri S., Shim K. Including Group-By in Query Optimization // Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann, San Mateo, USA, 1994. San Francisco: Morgan Kaufmann Publishers Inc., 1994. Pp. 354-366.

[7]. Khaitan P., Satish K. M., Korra S. B., Jena S. K. Improved query plans for unnesting nested SQL queries // Proceedings of 2nd International Conference on Computer Science and its Applications, December 10-12, South Korea, 2009. Jeju Island: IEEE, 2009. Pp. 147-152.

[8]. Muralikrishna M. Improved unnesting algorithms for join aggregate SQL queries // Proceedings of the 18th International Conference on Very Large Data Bases, August 23-27, Vancouver, Canada, 1992. San Francisco: Morgan Kaufmann Publishers Inc., 1992. Pp. 91-102.

[9]. Sippu S., Soisalon-Soininen E. An Analysis of Magic Sets and Related Optimization Strategies for Logic Queries // Journal of the ACM, Volume 43, № 6, November 1996.

[10]. Мендкович Н. А., Кузнецов С. Д. Обзор развития методов лексической оптимизации запросов // Труды Института системного программирования, т. 23, М., ИСП РАН, 2012. Сс. 204-207.
[11]. 7.2.1.2 How MySQL Optimizes WHERE Clauses [http://dev.mysql.com/doc/refman/5.5/en/where-optimizations.html]. [Обращение 20 ноября 2012].
[12]. MySQL 5.5 Reference Manual. Chapter 7. Optimization. http://dev.mysql.com/doc/refman/5.5/en/optimization.html, [Опубликовано в 2010 году, обращение 20 ноября 2012].
[13]. Пакет PostgreSQL 8.3.3. Адрес файла postgresql-8.3.3\src\backend\optimizer\util\predtest.c.
[14]. Query Optimization in Oracle Database 10g Release 2. An Oracle White Paper, June 2005. Redwood Shores, Oracle Corporation, 2005.
[15]. Mendkovich N., Kuznetcov S. New Algorithms for Lexical Query Optimization // Proceedings of the 31st International Conference on Information Technology Interfaces. Cavtat/Dubrovnik, Croatia, June 22-25, 2009. Zagreb: University of Zagreb, 2009. Pp. 187-192.
[16]. Кузнецов С. Д., Мендкович Н. А. Новые алгоритмы лексической оптимизации запросов // Модели и анализ информационных систем, 2009. Т. 16, № 4. С. 22-33.
[17]. Мендкович Н. А., Кузнецов С. Д. Оптимизация конъюнктов условий в составе запросов // Модели и анализ информационных систем, 2011. Т. 18, № 3. С. 144-154.
[18]. Hall P. A. V. Optimization of single expressions in a relational data base system // IBM Journal of Research and Development, 1976. Volume 20, Number 3.
[19]. Bellamkonda S., Ahmed R., Witkowski A., Amor A., Zait M., Lin Ch.-Ch. Enhanced Subquery Optimizations in Oracle / // Proceedings of the 35th international conference on Very large data base, August 2009. Volume 2 Issue 2. P. 1368.
[20]. Veitch E. W. A Chart Method for Simplifying Truth Functions // ACM Annual Conference/Annual Meeting: Proceedings of the 1952 ACM Annual Meeting, Pittsburg: ACM, NY, 1952.
[21]. Karnaugh M. The Map Method for Synthesis of Combinational Logic Circuits // Transactions of the American Institute of Electrical Engineers. Part I, № 72 (9), November 1953.
[22]. Quin W. V. O cores and prime implicants of truth functions // American Mathematics Monthly. 1959. V. 66. №9.
[23]. McCluskey E. J. Minimization of Boolean Functions // The Bell System Technical Journal. November 1956. V. 35, Issue 5.
[24]. Wu M. C. Query Optimization for Selecting Using Bitmaps // ACM SIGMOD Record Volume 28 Issue 2, June 1999. P. 234.
[25]. Тарасенко П. Ф., Бухарова М. Ф. Технология «The Reporter» для построения отчетов по базам данных // Вестник Томского Государственного Университета, № 275, апрель 2002.
[26]. Sarma A., Theobald M., Widom J. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. Technical Report. Stanford, 2007. [http://ilpubs.stanford.edu:8090/800/] [Обращение 20 ноября 2012].
[27]. Saad Y., van der Vorst H. A. Iterative solution of linear systems in the 20th Century // Journal of Computational and Applied Mathmatics, Issue 123, 2000.

[28]. Benzi M. The Early History of Matrix Iterations: With a Focus on the Italian Contribution // SIAM Conference on Applied Linear Algebra, Monterey Bay, Seaside, California, 26 October 2009.
[29]. Бронштейн И. Н., Семедяев К. А. Справочник по математике для инженеров и учащихся втузов. 13-е изд. исправленное. М.: Наука, 1986.
[30]. Ozsu M. N., Valduriez P. Principles of Distributed Database Systems. Second Edition. New Jersey, 1999. P. 233.
[31]. Graefe G. Query Evaluation Techniques for Large Databases // ACM Computing Surveys, 1993. Volume 25, Issue 2.
[32]. Hromkovi J. Theoretical Computer Science: Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography. Berlin: Springer, 2004.
[33]. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. Third Edition. Cambridge-London: Massachusetts Institute of Technology Press, 2009.

Minimization of data base query's conditions: evoluation of efficiency

Kuznetsov S.D.
ISP RAS, Moscow, Russia
kuzloc@ispras.ru
Mendkovich N.A.
OOO «FREEnet Group», Moscow, Russia
mend@f-group.ru

Abstract: This paper describes enhanced algorisms of lexical optimization query. These algorisms detect and remove redundant conditions from query restriction to simplify it. The paper also presents results of implementation of these optimization techniques and those effects on query processing speed. The paper includes four sections. The first section (Introduction) provides general context of the paper. The second section describes three proposed algorithms of lexical query optimization. The first one is the algorithm of absorption. This algorithm allows to find and remove a wide set of conditions that are redundant but are not equal textually even after standardization of whole restriction expression. The second algorithm is an adaptation of well-known Quin-McCluskey algorithm initially designed for minimization of Boolean functions. The last algorithm of lexical query optimization is based on techniques for optimization of systems of linear inequalities. The third section of the paper discusses results of efficiency evaluation of the proposed algorithms. The forth section concludes the paper.

Keywords: query optimization, lexical optimization, query processing

References:

[1]. Jarke M., Koch J. Query Optimization in Database Systems. ACM Computing Surveys (CSUR), 1984. March, Volume 16, Issue 2. Pp. 111-152.
[2]. Mannino M. V., Chu P., Sager T. Statistical profile estimation in database systems. ACM Computing Surveys, Volume 20, Issue 3. September 1988.
[3]. Ionnidis Y. E. Query Optimization. The Computer Science and Engineering Handbook. Boca Raton, CRC Press, 1996.
[4]. Chaudhari S. An Overview of Query Optimization in Relational Systems. Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems, 1998.
[5]. Kuznetsov S. D. Metody optimizacii vypolnenija zaprosov v reljacionnyh SUBD [Query optimization techniques for relational DBMSs]. [http://www.citforum.ru/database/articles/art_26.shtml] (in Russian).
[6]. Chaudhuri S., Shim K. Including Group-By in Query Optimization. Proceedings of the 20th International Conference on Very Large Data Bases, Morgan Kaufmann, San Mateo, USA, 1994. San Francisco: Morgan Kaufmann Publishers Inc., 1994. Pp. 354-366.
[7]. Khaitan P., Satish K. M., Korra S. B., Jena S. K. Improved query plans for unnesting nested SQL queries. Proceedings of 2nd International Conference on Computer Science

and its Applications, December 10-12, South Korea, 2009. Jeju Island: IEEE, 2009. Pp. 147-152.
[8]. Muralikrishna M. Improved unnesting algorithms for join aggregate SQL queries. Proceedings of the 18th International Conference on Very Large Data Bases, August 23-27, Vancouver, Canada, 1992. San Francisco: Morgan Kaufmann Publishers Inc., 1992. Pp. 91-102.
[9]. Sippu S., Soisalon-Soininen E. An Analysis of Magic Sets and Related Optimization Strategies for Logic Queries. Journal of the ACM, Volume 43, № 6, November 1996.
[10]. Mendkovich N. A., Kuznetsov S. D. Obzor razvitija metodov leksicheskoj optimizacii zaprosov [An Overwiew of Evolution of Lexical Query Optimization Techniques]. Trudy Instituta sistemnogo programmirovanija, t. 23, M., ISP RAN, 2012. Ss. 204-207 (in Russian).
[11]. 7.2.1.2 How MySQL Optimizes WHERE Clauses [http://dev.mysql.com/doc/refman/5.5/en/where-optimizations.html.
[12]. MySQL 5.5 Reference Manual. Chapter 7. Optimization. http://dev.mysql.com/doc/refman/5.5/en/optimization.html.
[13]. Paket PostgreSQL 8.3.3. Adres fajla postgresql-8.3.3\src\backend\optimizer\util\predtest.c.
[14]. Query Optimization in Oracle Database 10g Release 2. An Oracle White Paper, June 2005. Redwood Shores, Oracle Corporation, 2005.
[15]. Mendkovich N., Kuznetcov S. New Algorithms for Lexical Query Optimization. Proceedings of the 31st International Conference on Information Technology Interfaces. Cavtat/Dubrovnik, Croatia, June 22-25, 2009. Zagreb: University of Zagreb, 2009. Pp. 187-192.
[16]. Kuznetsov S. D., Mendkovich N. A. Novye algoritmy leksicheskoj optimizacii zaprosov New [Algorithms for Lexical Query Optimization]. Modeli i analiz informacionnyh sistem, 2009. T. 16, № 4. S. 22-33 (in Russian).
[17]. Mendkovich N. A., Kuznetsov S. D. Optimizacija kon#junktov uslovij v sostave zaprosov [Optimization of query condition' conjucts]. Modeli i analiz informacionnyh sistem, 2011. T. 18, № 3. S. 144-154 (in Russian).
[18]. Hall P. A. V. Optimization of single expressions in a relational data base system. IBM Journal of Research and Development, 1976. Volume 20, Number 3.
[19]. Bellamkonda S., Ahmed R., Witkowski A., Amor A., Zait M., Lin Ch.-Ch. Enhanced Subquery Optimizations in Oracle /. Proceedings of the 35th international conference on Very large data base, August 2009. Volume 2 Issue 2. P. 1368.
[20]. Veitch E. W. A Chart Method for Simplifying Truth Functions. ACM Annual Conference/Annual Meeting: Proceedings of the 1952 ACM Annual Meeting, Pittsburg: ACM, NY, 1952.
[21]. Karnaugh M. The Map Method for Synthesis of Combinational Logic Circuits. Transactions of the American Institute of Electrical Engineers. Part I, № 72 (9), November 1953.
[22]. Quin W. V. O cores and prime implicants of truth functions. American Mathematics Monthly. 1959. V. 66. №9.
[23]. McCluskey E. J. Minimization of Boolean Functions. The Bell System Technical Journal. November 1956. V. 35, Issue 5.
[24]. Wu M. C. Query Optimization for Selecting Using Bitmaps. ACM SIGMOD Record Volume 28 Issue 2, June 1999. P. 234.

[25]. Tarasenko P. F., Buharova M. F. Tehnologija «The Reporter» dlja postroenija otchetov po bazam dannyh. Vestnik Tomskogo Gosudarstvennogo Universiteta, № 275, aprel' 2002 (in Russian).
[26]. Sarma A., Theobald M., Widom J. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. Technical Report. Stanford, 2007. [http://ilpubs.stanford.edu:8090/800/] [Обращение 20 ноября 2012].
[27]. Saad Y., van der Vorst H. A. Iterative solution of linear systems in the 20th Century. Journal of Computational and Applied Mathmatics, Issue 123, 2000.
[28]. Benzi M. The Early History of Matrix Iterations: With a Focus on the Italian Contribution. SIAM Conference on Applied Linear Algebra, Monterey Bay, Seaside, California, 26 October 2009.
[29]. Bronshtejn I. N., Semedjaev K. A. Spravochnik po matematike dlja inzhenerov i uchashhihsja vtuzov [Directory by mathematics for engineers and students of technical schools]. 13-e izd. ispravlennoe. M.: Nauka, 1986 (in Russian).
[30]. Ozsu M. N., Valduriez P. Principles of Distributed Database Systems. Second Edition. New Jersey, 1999. P. 233.
[31]. Graefe G. Query Evaluation Techniques for Large Databases. ACM Computing Surveys, 1993. Volume 25, Issue 2.
[32]. Hromkovi J. Theoretical Computer Science: Introduction to Automata, Computability, Complexity, Algorithmics, Randomization, Communication, and Cryptography. Berlin: Springer, 2004.
[33]. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms. Third Edition. Cambridge-London: Massachusetts Institute of Technology Press, 2009.

Исследование и развитие метода декомпозиции для анализа больших пространственных данных [1]

Золотов В.А., Семенов В.А.

Аннотация. Статья посвящена развитию метода декомпозиции для индексации, поиска и анализа больших пространственных данных. Главное внимание уделяется алгоритмам, основанным на регулярных октальных деревьях и обеспечивающим эффективное решение ряда вычислительных задач. Исследуемые алгоритмы определения столкновений и выборки по заданной области, в частности, применимы для моделирования сложных динамических пространственно-трехмерных сцен с объектами, имеющими протяженные границы. Для модельного набора данных на основе вероятностного анализа выводятся оценки сложности, которые обобщают и улучшают известные результаты, а также служат теоретическим обоснованием для применения алгоритмов к более широкому классу приложений.

Ключевые слова: Пространственные индексы, октальные деревья, вычислительная сложность.

1. Введение

Стремительный рост объемов информации, а также необходимость ее анализа приводят к развитию новых подходов к управлению данными и, в частности, методов пространственного индексирования, без которых невозможен быстрый поиск и обработка в геоинформационных базах данных, системах логистического обеспечения, системах автоматизации проектирования, системах управления проектами. Как правило, популярные универсальные и специализированные СУБД предусматривают для этих целей средства пространственного индексирования и поиска. Подобные средства успешно справляются с обработкой статической информации, однако часто не приспособлены для данных, подлежащих перманентным изменениям [1]. Проблемы эффективного поиска и анализа еще более усложняются, когда информация представляет собой не просто массивы точек, а сложно-структурированные наборы данных, например, множества объектов с протяженными пространственными границами [2]. Класс подобных

[1] Исследование поддержано грантом РФФИ 13-07-00294.

приложений чрезвычайно широк и охватывает не только перечисленные выше прикладные области, но и многочисленные системы компьютерной графики, визуализации и анимации. Эти факторы определяют актуальность темы и огромный интерес, как со стороны научного сообщества, так и производителей системного и прикладного программного обеспечения. В частности, активные разработки в этой области ведут компании Google, Oracle, IBM, Autodesk, Bentley, Intergraph, AVEVA, сталкивающиеся с проблемой увеличения объемов анализируемой пространственно-временной информации.

В работе [3] были системно проанализированы фундаментальные семейства методов индексации и поиска многомерных данных. В частности, рассматривались структуры поиска на интервалах, сбалансированные ветвистые деревья во внешней памяти, бинарные деревья пространственной декомпозиции, префиксные деревья, нерегулярные и регулярные многоуровневые сетки, метрические деревья, а также связанные с ними разнообразные методики хэширования, расщепления и кластеризации.

Большое внимание было уделено методам пространственной индексации на основе регулярных октальных деревьев. Главным их достоинством является простота развертывания и модификации индексов, обусловленная априори известным положением секущих плоскостей. Это позволяет относительно легко обновлять индексы и обеспечивать эффективность исполнения типовых пространственных запросов при перманентных изменениях самих данных. Однако данные методы не обеспечивают сбалансированность структур индексов и могут приводить к их деградации в тех случаях, когда данные неравномерно распределены по пространству или имеют протяженные границы. Известные оценки сложности в подобных наихудших случаях приводят к пессимистическим выводам и не определяют реальных границ применимости методов декомпозиции на основе регулярных октальных деревьев. По-видимому, они нуждаются в более детальном исследовании и развитии, исходя из оценок сложности в среднем, которые могут быть получены на основе вероятностного анализа основных алгоритмов при необходимой конкретизации условий прикладных задач.

В разделе 2 рассматривается постановка задачи, связанной с визуальным анализом пространственно-трехмерных сцен. В рамках данной постановки исследуется алгоритм развертывания регулярных октальных деревьев. В разделе 3 описываются алгоритмы определения столкновений и выборки объектов по заданной области, основанные на регулярных октальных деревьях. Полученные для них оценки сложности в среднем обобщают и существенно улучшают известные результаты, а выработанные рекомендации создают предпосылки к широкому практическому применению методов декомпозиции для индексации, поиска и анализа больших пространственных данных. В заключении подводятся итоги проведенного исследования.

2. Пространственная декомпозиция на основе октальных деревьев

Начнем с анализа классического метода пространственной декомпозиции, основанного на регулярных октальных деревьях. В трехмерном случае параллелепипед, пространственно ограничивающий весь набор данных (или в англоязычной литературе AABB — Axis Aligned Bounding Box), разбивается на восемь равных частей плоскостями, перпендикулярными каждой из координатных осей. Процесс рекурсивно применяется до тех пор, пока количество элементов данных в каждом вновь образованном октанте не окажется ниже некоторого предустановленного порога $m > 1$. С результирующим пространственным разбиением ассоциируется соответствующее дерево. Корень дерева соответствует исходному параллелепипеду, содержащему в себе весь набор данных, а вершины — вложенным октантам, группирующим данные на разных иерархических уровнях пространственной декомпозиции. Пример декомпозиции и полученного октального дерева приведен на рис. 1.

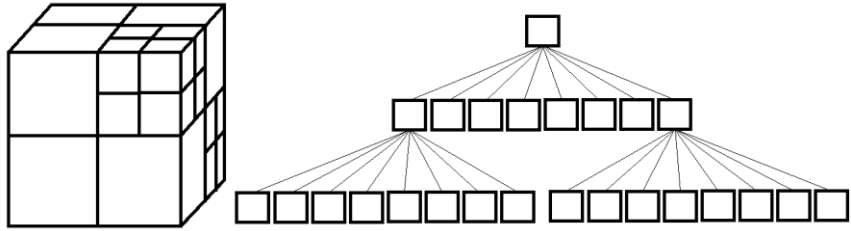


Рисунок 1. Пример пространственной декомпозиции и соответствующего ей октального дерева.

Заметим, что способ ассоциирования элементов данных с вершинами октального дерева, в конечном счете, определяется способом их дальнейшей локализации и поэтому зависит от их пространственной семантики. Точечные данные могут быть непосредственно ассоциированы с листовыми октантами. Однако геометрические объекты с протяженными границами могут занимать определенный объем исходного параллелепипеда. Поэтому более рациональным представляется их ассоциирование с теми октантами, в которых они могут быть размещены полностью [4]. Для такого способа есть и иные причины, связанные с возможным пересечением секущими плоскостями даже небольших объектов и невозможностью установить простое соответствие между ними и листовыми октантами.

Рассмотрим псевдокод алгоритма построения октального дерева, приведенный ниже:

FUNCTION BUILD_OCTREE(box, objects, m)
BOX box
OBJECT COLLECTION objects

```
INTEGER m
RETURNS POINTER NODE root_octant

root_octant = new NODE(box);

FOR_EACH (POINTER OBJECT o IN objects)
        INSERT(o, root_octant)


PROCEDURE INSERT(object,octant)
POINTER OBJECT object
POINTER NODE octant

POINTER NODE child = LOCALIZE(object, octant)
IF(child)
        RETURN INSERT(object,child)
ELSE
        ADD(object,GET_OBJECTS(octant))

IF(CAN_SPLIT(octant))
        SPLIT(octant)


PROCEDURE SPLIT(octant)
POINTER NODE octant

DECOMPOSE(octant)
FOR_EACH(OBJECT POINTER object IN GET_OBJECTS(octant))
        {
        POINTER NODE child = LOCALIZE(object,octant)
        if(child)
                {
                ERASE(object,GET_OBJECTS(octant))
                INSERT(object, child)
                }
        }
```

Алгоритм 1. Построение регулярного октального дерева.

Функция BUILD_OCTREE принимает в качестве входных параметров ограничивающий параллелепипед box, множество объектов (элементов данных с пространственной семантикой) $objects$, а также параметр заполнения вершин октального дерева m. Возвращаемое значение — указатель на корень развернутого октального дерева. Для построения используется вспомогательная процедура INSERT, которая выполняет вставку

заданного объекта $object \in objects$ в октальное дерево, начиная с заданного родительского октанта $octant$. Остановимся на этой процедуре более подробно. В псевдокоде процедуры используется вспомогательная функция LOCALIZE, которая определяет, в каком из дочерних октантов может быть полностью размещен объект. Если исходный октант является листовым или объект не локализуется ни в одном из дочерних октантов, функция возвращает нулевое значение. При успешной локализации процедура вставки рекурсивно повторяется. В противном случае, посредством процедуры ADD, объект приписывается исходному октанту (более точно, коллекции объектов, получаемой при помощи вспомогательной функции GET_OBJECTS). Функция CAN_SPLIT проверяет условие наполнения октанта и целесообразность его разбиения, которое реализуется при помощи процедуры SPLIT. При вызове процедуры DECOMPOSE создается восемь дочерних октантов и предпринимается попытка перераспределить по ним объекты родительского октанта. Перенос объектов в дочерние октанты осуществляется посредством процедур INSERT и ERASE.

Определение. Октальное дерево, построенное на основе рассмотренного алгоритма 1 с порогом наполнения октантов m, назовем регулярным и обозначим $Octree(m)$.

Регулярное октальное дерево может использоваться для решения различного рода вычислительных задач, связанных с поиском и анализом пространственно-временных данных. Естественно, что затраты на развертывание дерева должны сполна компенсироваться более быстрым решением целевых задач. Оценим затраты на построение октального дерева. Заметим, что подобные оценки могут существенно зависеть от специфики прикладных данных и анализ наихудшего случая, как правило, приводит к довольно пессимистическим результатам, не отражающим реальные показатели производительности для большинства приложений. Поэтому получим оценки сложности в среднем на основе вероятностного анализа некоторого упрощенного набора данных.

Для этого рассмотрим набор данных, связанный с визуальным анализом пространственно-трехмерных сцен и допускающий содержательную параметризацию. Пусть сцена представляется набором n трехмерных геометрических объектов, равномерно расположенных внутри единичного куба. Ограничивающим объемом каждого объекта является куб с характерным размером сторон $0 \leq l \leq 1$. В рамках подобной постановки x, y, z-координаты центров являются независимыми равномерно распределенными величинами на соответствующих отрезках $[l, 1-l]$. Для обсуждаемых задач анализа сцен будет применяться единая техника пространственной локализации объектов на основе их ограничивающих объемов, поэтому в дальнейшем мы не делаем никаких различий между понятиями геометрического объекта и его ограничивающего параллелепипеда. Рисунок 2а иллюстрирует область возможного расположения центров объектов внутри объема всей сцены. При

этом величина $\rho(n,l)=\frac{nl^3}{(1-l)^3}$ определяет эффективную пространственную плотность данных.

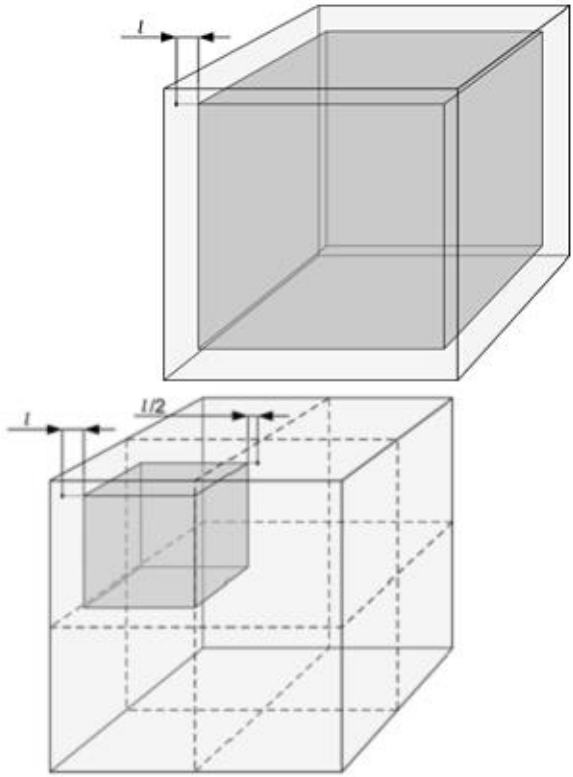


Рисунок 2. а) Область возможного расположения центров объектов внутри ограничивающего объема набора данных. б) Область возможного расположения центров объектов, локализуемых в октантах следующего уровня.

Определение. Набор n кубов назовем модельным и обозначим как $S(n,l)$, если они равномерно распределены в единичном кубе, а их ребра ориентированы вдоль главных координатных осей и имеют относительный размер $0 \le l < \frac{1}{2}$.

Заметим, что при относительном размере объектов $l \ge \frac{1}{2}$ октальная структура вырождается, поскольку все объекты локализуются в самом верхнем октанте при глубине дерева $h=1$. Поэтому при дальнейшем рассмотрении мы ограничимся содержательным случаем $l < \frac{1}{2}$. Исследуем свойства регулярных октальных деревьев, развернутых для модельного набора данных. Для этого сформулируем и докажем несколько вспомогательных утверждений.

Лемма 1. Пусть регулярное октальное дерево глубиной $h \le \left\lceil \log_2 \frac{1}{l} \right\rceil$ построено для модельного набора $S(n, l)$. Тогда математическое ожидание числа объектов в октантах i-го уровня и ниже ($1 \le i \le h$) есть

$$n_i = n \frac{(1 - 2^{i-1} l)^3}{(1 - l)^3}$$

Доказательство:

Прежде всего, заметим, что глубина регулярного октального дерева всегда ограничена величиной $\left\lceil \log_2 \frac{1}{l} \right\rceil$, поскольку объекты модельного набора не могут быть локализованы в октантах размера равного или меньшего, чем l. Поэтому анализу подлежат лишь допустимые уровни локализации объектов $1 \le i \le \left\lceil \log_2 \frac{1}{l} \right\rceil$.

Далее, уточним область трехмерного пространства, где могут располагаться центры объектов, приписанных октантам разных уровней. На рисунке 2а темным цветом выделена область возможного расположения центров объектов в октанте верхнего уровня, поскольку они не могут находиться к граням ограничивающего параллелепипеда ближе, чем на расстоянии $\frac{l}{2}$. По построению дерева объект приписывается родительскому октанту, если он пересекается одной из секущих плоскостей. В противном случае предпринимается попытка локализовать его в одном из дочерних октантов следующего уровня. На рисунке 1б темным цветом показана одна из таких областей в октантах второго уровня. Чтобы оценить число объектов, локализуемых на втором уровне дерева или ниже, воспользуемся предположением об их равномерном пространственном распределении и геометрической интерпретацией вероятности.

Вероятность того, что объект не будет пересечен секущими верхнего октанта и, следовательно, будет локализован на втором уровне дерева или ниже, равна

$$P_2^1 = \frac{(1 - 2l)^3}{(1 - l)^3}$$

В знаменателе выражения — объем области допустимого расположения центров объектов в октанте первого уровня, в числителе — суммарный объем областей возможного расположения центров объектов, локализуемых в восьми дочерних октантах. Рассуждая аналогично для оставшихся уровней, получим вероятность того, что объект не будет пересечен секущими i-го октанта и будет локализован на следующих уровнях дерева

$$P_{i+1}^{i} = \frac{(\frac{1}{2^{i-1}} - 2l)^3}{(\frac{1}{2^{i-1}} - l)^3} = \frac{(1 - 2^i l)^3}{(1 - 2^{i-1} l)^3}$$

Тогда математическое ожидание числа объектов, локализованных в октантах *i*-го уровня и ниже ($1 \le i \le h$) выражается как

$$n_i = n \cdot P_2^1 \cdot P_3^2 \cdot P_4^3 \cdot \ldots \cdot P_{i-1}^{i-2} \cdot P_i^{i-1}$$

или

$$n_i = n \frac{(1-2l)^3}{(1-l)^3} \cdot \frac{(1-4l)^3}{(1-2l)^3} \cdot \frac{(1-8l)^3}{(1-4l)^3} \cdot \ldots \cdot \frac{(1-2^{i-1}l)^3}{(1-2^{i-2}l)^3}$$

$$= n \frac{(1-2^{i-1}l)^3}{(1-l)^3}$$

Что и требовалось доказать.

Лемма 2. Пусть регулярное октальное дерево глубиной $h \le \left\lfloor \log_2 \frac{1}{l} \right\rfloor$ построено для модельного набора $S(n, l)$, тогда математическое ожидание числа объектов в октанте *i*-го уровня есть

$$N_i = \begin{cases} n \frac{(1-2^{i-1}l)^3 - (1-2^i l)^3}{8^{i-1}(1-l)^3}, 1 \le i < h \\ n \frac{(1-2^{i-1}l)^3}{8^{i-1}(1-l)^3}, i = h \end{cases}$$

Доказательство:

Лемма 1 определяет ожидаемое число объектов в октантах *i*-го уровня и ниже как n_i, а ожидаемое число объектов в октантах *i+1*-го уровня и ниже как n_{i+1}. Поскольку на *i*-ом уровне дерева имеется 8^{i-1} октантов, то математическое ожидание числа объектов в одном таком октанте определяется выражением

$$N_i = \frac{1}{8^{i-1}}(n_i - n_{i+1}) = n \frac{(1-2^{i-1}l)^3 - (1-2^i l)^3}{8^{i-1}(1-l)^3}$$

Выражение справедливо для всех октантов уровней $1 \le i < h$. Для определения ожидаемого числа объектов в листовых октантах воспользуемся непосредственно леммой 2. Тогда для октантов уровня $i = h$ имеем

$$N_i = n \frac{(1-2^{i-1}l)^3}{8^{i-1}(1-l)^3}$$

Что и требовалось доказать.

На основе доказанных лемм удается оценить глубину регулярного октального дерева. Имеет место следующая теорема.

Теорема 1. Ожидаемая глубина регулярного октального дерева $Octree(m)$, построенного для модельного набора данных $S(n,l)$ с плотностью пространственного заполнения $\rho(n,l)$, определяется следующим выражением:

$$h = \begin{cases} \left\lceil \log_2 \dfrac{2}{l + (1-l)\sqrt[3]{\dfrac{m}{n}}} \right\rceil, & m \geq \rho(n,l) \\ \left\lceil \log_2 \dfrac{1}{l} \right\rceil, & m < \rho(n,l) \end{cases}$$

Доказательство:

Согласно алгоритму построения регулярного октального дерева его глубина ограничивается следующими факторами:

— размер листового октанта должен быть строго больше габаритов объекта l, чтобы объект мог быть размещен в нем целиком;

— разбиение октантов возможно лишь при их достаточном наполнении, превышающем заданный порог m.

Чтобы процедура разбиения могла быть применена к октантам i-го уровня в соответствии с леммой 2, должны выполняться следующие условия:

$$\begin{cases} \dfrac{1}{2^{i-1}} > 2l \\ n\dfrac{(1-2^{i-1}l)^3}{8^{i-1}(1-l)^3} > m \end{cases}$$

В листовых октантах, по крайней мере, одно из приведенных условий не выполняется. Невыполнение первого условия приводит к неравенству $\frac{1}{2^{h-1}} \leq 2l$, а в комбинации с ранее полученной оценкой $h \leq \left\lceil \log_2 \frac{1}{l} \right\rceil$, имеем для глубины дерева строгое равенство $h_1 = \left\lceil \log_2 \frac{1}{l} \right\rceil$.

Рассмотрим более подробно второе условие. Условия разбиения должны выполняться в октантах предпоследнего уровня и не должны выполняться в листовых октантах с ожидаемым числом объектов, не превышающим заданный порог m. Следовательно, имеют место следующие неравенства:

$$n\frac{(1-2^{h-1}l)^3}{8^{h-1}(1-l)^3} \leq m < n\frac{(1-2^{h-2}l)^3}{8^{h-2}(1-l)^3}$$

Их согласованный анализ приводит к следующей оценке глубины дерева:

$$h_2 = \left\lceil \log_2 \frac{2}{l + (1-l)\sqrt[3]{\frac{m}{n}}} \right\rceil$$

Поскольку рост дерева прекращается, когда нарушается одно из условий разбиения октантов, то ожидаемая глубина регулярного октального дерева будет определяться минимальным из найденных значений $h = \min(h_1, h_2)$.

Сравнив найденные значения, получим соотношение параметров, при котором глубина определяется первым фактором. В самом деле, из условия $h_1 \le h_2$ следуют соотношения $\log_2 \frac{2}{l+(1-l)\sqrt[3]{\frac{m}{n}}} \le \log_2 \frac{1}{l}$, $l \le (1-l)\sqrt[3]{\frac{m}{n}}$, $m \ge \frac{nl^3}{(1-l)^3}$ или $m \ge \rho(n,l)$ с учетом определения эффективной пространственной плотности данных. Тем самым, имеет место $h = \begin{cases} h_1, & m \ge \rho(n,l) \\ h_2, & m < \rho(n,l) \end{cases}$ и теорема доказана.

3. Теоретический анализ сложности метода декомпозиции

Итак, проанализируем вычислительные затраты на развертывание регулярного октального дерева, а также на выполнение запросов, связанных с выборкой объектов по заданной пространственной области и определением столкновений.

Как отмечалось, асимптотические оценки сложности, а также оценки сложности в наихудшем случае не отражают реальную эффективность метода декомпозиции для многих важных классов приложений. Поэтому воспользуемся описанным модельным набором данных и получим оценку сложности на основе проведенного вероятностного анализа и полученных результатов о свойствах октальных деревьев. Существенной стороной проводимого анализа является попытка получить оценки не в асимптотическом приближении, а на всем диапазоне изменения размерности и других параметров задачи. При анализе сложности будем учитывать только две, наиболее часто используемые вычислительные операции, а именно: определение принадлежности объекта одному из дочерних октантов с условной стоимостью C_L и определение факта пересечения двух объектов со стоимостью C_I в том же самом предположении, что объекты геометрически представляются своими ограничивающими параллелепипедами. Затраты на конструирование новых октантов в структуре дерева, а также затраты на вставку и удаление ассоциированных объектов исключаются из рассмотрения. На самом деле эти операции всегда применяются в контексте основных вычислительных операций и поэтому неявно могут учитываться уже введенными константами.

3.1 Анализ стоимости развертывания регулярного октального дерева

Имеет место следующая лемма.

Лемма 3. Средняя стоимость развертывания регулярного октального дерева глубиной h для модельного набора данных $S(n,l)$ есть

$$Q_{deploy} = \frac{C_L n}{56(1-l)^3}(56h - 56 + 168l - 84 \cdot 2^h l - 56l^2 + 14 \cdot 2^{2h} l^2 + 8l^3 - 2^{3h} l^3)$$

Доказательство:

По лемме 1 математическое ожидание числа объектов в октантах i-го уровня и ниже ($1 \le i \le h$) есть

$$n_i = n\frac{(1-2^{i-1}l)^3}{(1-l)^3}$$

Именно столько объектов необходимо проанализировать для того, чтобы принять решение об их локализации на уровне i. Анализу не подлежат объекты, приписанные листовым октантам, поскольку необходимость их перераспределения в дочерних октантах отсутствует. Исходя из этих рассуждений, стоимость построения октального дерева определяется суммой затрат на локализацию объектов на уровнях $1,2,\dots,h-1$ и определяется следующей формулой:

$$Q_{deploy} = \sum_{i=1}^{h-1} C_L n\frac{(1-2^{i-1}l)^3}{(1-l)^3} = \frac{C_L n}{(1-l)^3}\sum_{i=1}^{h-1}(1-2^{i-1}l)^3$$

$$= \frac{C_L n}{56(1-l)^3}(56h - 56 + 168l - 84 \cdot 2^h l - 56l^2 + 14 \cdot 2^{2h} l^2 + 8l^3 - 2^{3h} l^3)$$

Лемма доказана.

Необходимо отметить, что в случае точечных данных ($l = 0$) стоимость построения октального дерева определяется выражением $Q_{deploy} = C_L n(h-1)$, что соответствует случаю полной локализации данных в листовых октантах и хорошо согласуется с известными результатами [5]. Заметим, что в силу леммы 1 количество объектов, подлежащих локализации, монотонно уменьшается с каждым уровнем и поэтому затраты на построение дерева всегда могут быть оценены как $Q_{deploy} \le C_L n(h-1)$ с учетом выражения для ожидаемой глубины дерева, основанного на результатах теоремы 1.

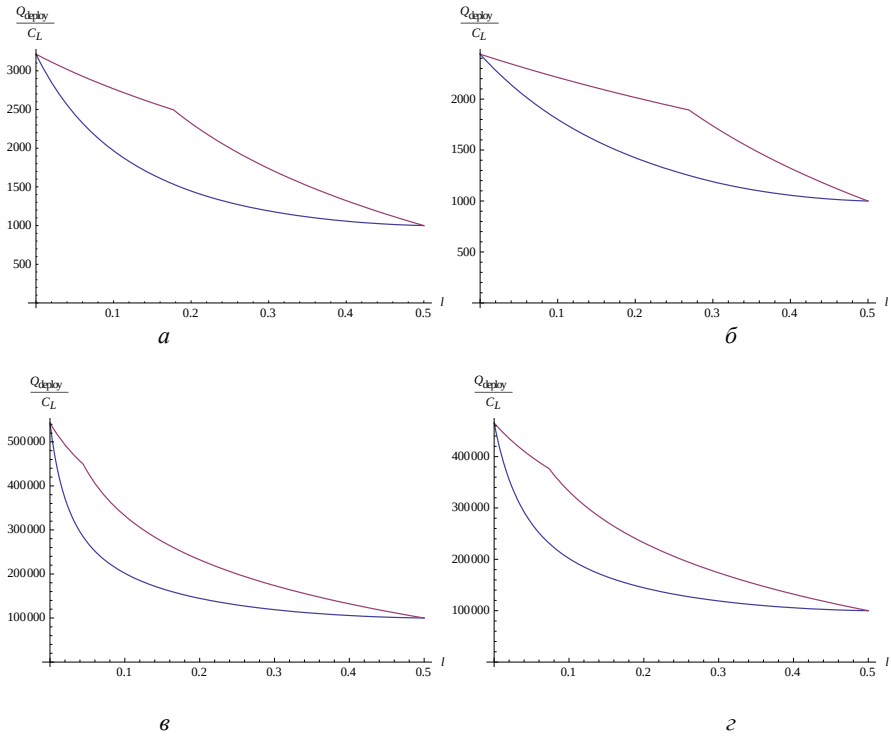


Рисунок 3. Зависимость стоимости развертывания дерева от габаритов объектов при разных соотношениях n и m. а) $n = 1000,\ m = 10$ *б)* $n = 1000,\ m = 50$ *в)* $n = 100000,\ m = 10$ *г)* $n = 100000,\ m = 50$

На рисунке 3 показаны кривые сложности развертывания октального дерева от габаритов объектов при разном соотношении общего числа объектов и порога наполненности октантов. Верхние кривые соответствуют приведенной приближенной оценке, нижние — воспроизводят функцию стоимости, полученную точно в предположениях леммы 3. Построенные графики показывают, что приближенная оценка отражает общий характер исследуемых зависимостей и отличается от функции стоимости на всем интервале изменения параметров не более, чем в два раза. Это дает основания применять ее на практике.

Теорема 2. Для модельного набора данных $S(n, l)$ регулярное октальное дерево $Octree(m)$ может быть построено в среднем за $Q_{deploy} = C_L q(n, m, l) \cdot n$, причем коэффициент $q(n, m, l)$ удовлетворяет следующим соотношениям:

$$q(n, m, l) = \left\lceil \frac{1}{3}\log_2 n - \frac{1}{3}\log_2 m \right\rceil \text{ при } l = 0,$$

$$q(n,m,l) \le \log_2 \frac{1}{l} \text{ при } l > 0$$

Доказательство:

Рассмотрим случай $l = 0$. По лемме 3

$$Q_{deploy} = \frac{C_L n}{56(1-l)^3}(56h - 56 + 168l - 84 \cdot 2^h l - 56l^2 + 14 \cdot 2^{2h} l^2 + 8l^3 - 2^{3h} l^3)$$

Подставляя $l = 0$, получим

$$Q_{deploy} = C_L n(h-1)$$

По теореме 1, при $l = 0$ выполняется следующее равенство:

$$h = \left\lceil \log_2 \frac{2}{l + (1-l)\sqrt[3]{\frac{m}{n}}} \right\rceil = \left\lceil \log_2 \frac{2}{\sqrt[3]{\frac{m}{n}}} \right\rceil = \left\lceil \log_2 \sqrt[3]{\frac{n}{m}} + 1 \right\rceil$$

$$= \left\lceil \frac{1}{3}\log_2 n - \frac{1}{3}\log_2 m \right\rceil + 1$$

Следовательно,

$$Q_{deploy} = C_L \left(\left\lceil \frac{1}{3}\log_2 n - \frac{1}{3}\log_2 m \right\rceil \right) \cdot n$$

Первая часть теоремы доказана.

Рассмотрим случай $l > 0$. Как было показано выше, имеет место общая оценка сложности развертывания дерева $Q_{deploy} \le C_L n(h-1)$. По теореме 1 глубина дерева всегда может быть ограничена как $h \le \log_2 \frac{2}{l}$. Следовательно, $Q_{deploy} \le C_L \log_2 \frac{1}{l} \cdot n$ и теорема доказана.

Теорема 2 имеет важные следствия, связанные с разным асимптотическим ростом сложности развертывания октального дерева для точечных и протяженных данных. Для первых ($l = 0$) сложность оценивается как $O(n\log_2 n)$, а для вторых ($l > 0$) — как $O(n)$. На рисунке 4 приведены графики сложности построения октального дерева от количества объектов в модельном наборе n при разных значениях порога наполненности октантов m, построенные на основе результатов леммы 3 для точечных (верхние кривые) и протяженных объектов (нижние кривые).

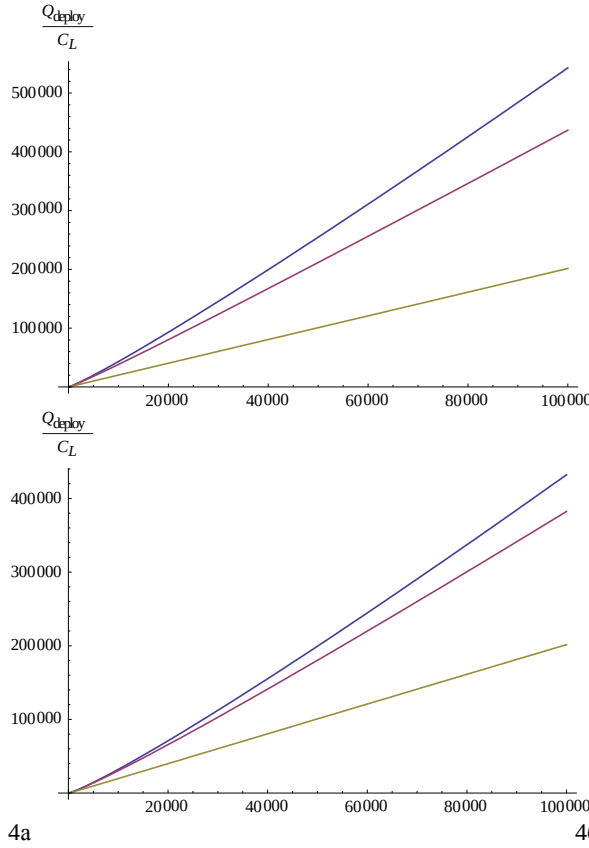


Рисунок 4. Зависимость стоимости построения октального дерева от количества объектов при различных габаритах объектов и параметра заполнения листовых октантов. а) $m = 10, l = 0{,}0.001{,}0.1$ *б)* $m = 100, l = 0{,}0.01{,}0.1$

Примечательно, что в исследуемом диапазоне размерности задачи (до ста тысяч объектов) все кривые выглядят схожим образом, несмотря на наличие логарифмического фактора в асимптотической оценке сложности для точечных объектов. Согласно приведенным графикам сложность построения дерева для протяженных объектов несколько ниже, чем для точечных данных. Это объясняется тем обстоятельством, что при одинаковом общем количестве протяженные объекты локализуются на более высоких уровнях октального дерева и не возникает необходимости в его дальнейшем развертывании.

3.2 Анализ сложности определения столкновений

Перейдем к оценкам затрат на выполнение типовых запросов, связанных с поиском столкновений объектов в пространственно-трехмерных сценах. Как

правило, точное пересечение двух геометрических объектов сложной формы требует большого объема вычислений. Поэтому более эффективной стратегией является предварительная локализация потенциальных столкновений с помощью простых тестов, основанных на сепарации пространства и пересечении ограничивающих объемов [6]. Точная процедура определения столкновений применяется лишь в случае положительного вердикта, что происходит в приложениях относительно редко и, как результат, существенно уменьшает общее время поиска столкновений. Эффект особенно ощутим для сцен, объекты которых геометрически представлены сложными аналитическими кривыми, поверхностями или полиэдрами с большим числом граней [7, 8].

Обсудим алгоритмические варианты использования октального дерева для локализации столкновений в сценах и получим оценки сложности в предположении, что сцена представлена модельным набором данных $S(n, l)$. Предположим, что октальное дерево развернуто и предстоит выявить все пары объектов, допускающие столкновения. В наивном алгоритме пришлось бы попарно пересечь все ограничивающие объемы объектов, что привело бы к затратам $\frac{1}{2} C_l n(n-1)$. Однако октальное дерево позволяет сделать это гораздо эффективнее с применением следующего алгоритма.

```
PROCEDURE CLASH (octant, result)
POINTER NODE octant
POINTER OBJECT PAIR COLLECTION result
FOR_EACH(OBJECT object1 IN OBJECTS(octant))
        {
        FOR_EACH(OBJECT object2 != object1 IN OBJECTS(octant))
                IF(IS_INTERSECTING(object1, object2))
                        INSERT_RESULT(result, PAIR(object1,
object2))

        FOR_EACH(NODE child IN CHILDREN(octant))
                CLASHCHILDREN(child, object1, result)
        }
FOR_EACH(NODE child IN CHILDREN(octant))
        CLASH(child, result)

PROCEDURE CLASHCHILDREN(octant, object, result)
POINTER NODE octant
POINTER OBJECT object
POINTER OBJECT PAIR COLLECTION result

IF(!IS_INTERSECTING(octant,object))
        RETURN
```

```
FOR_EACH(OBJECT o IN OBJECTS(octant))
        IF(IS_INTERSECTING(object, o))
                INSERT_RESULT(PAIR(object, o), result)

FOR_EACH(NODE child IN CHILDREN(octant))
        CLASHCHILDREN(child, object, result)
```

Алгоритм 2. Поиск столкновений при помощи регулярного октального дерева.

Процедура CLASH принимает в качестве параметра указатель на текущий октант, а также указатель на коллекцию результатов, каждый из которых представлен в виде пары объектов, имеющих потенциальную коллизию. В теле этой процедуры происходит поиск коллизий среди объектов, принадлежащих текущему октанту, а затем проверяются пересечения этих объектов с объектами, принадлежащими дочерним октантам при помощи вспомогательной процедуры CLASHCHILDREN, принимающей в качестве параметра объект и, посредствам поиска в глубину, ищущей все объекты, имеющие пересечения с данным. Затем процедура CLASH рекурсивно повторяется для всех дочерних октантов текущего октанта.

Перейдем к оценкам сложности для описанного алгоритма.

Лемма 4. Пусть регулярное октальное дерево $Octree(m)$ глубиной h развернуто для модельной сцены $S(n, l)$. Тогда столкновения объектов в ней могут быть локализованы за Q_{clash} в соответствии со следующей оценкой:

$$Q_{clash} \le \frac{C_I n^2}{2 \cdot 8^h (1-l)^6}(8 + l(357 \cdot 16^h l^3 - 28 \cdot 32^h l^4 + 64^h l^5 - 128(3 - 3l + l^2) + 56 \cdot 2^h\big(3 + 8l(3 - 3l + l^2)\big) - 42 \cdot 4^h l\big(23 + 16l(3 - 3l + l^2)\big) + 8^h l(114 + 1970l - 1483l^2 + 106l^3 + 105l^4)) + 14h(45 + 8l(3 - 3l + l^2))))))$$

Анализ совпадений точечных данных ($l = 0$) в сцене может быть проведен за $Q_{clash} \le \frac{C_I nm}{2}$.

Доказательство:

Описанный выше алгоритм 2 предполагает последовательный обход октантов всех уровней, начиная с самого верхнего. При обходе очередного октанта сначала анализируются возможные парные столкновения объектов внутри него. Для каждой ячейки это потребует $\frac{1}{2} C_I n_i (n_i - 1) \le \frac{C_I {n_i}^2}{2}$. Затем анализируются возможные столкновения внутренних объектов рассматриваемого октанта с объектами, приписанными дочерним октантам.

Заметим, что каждый внутренний объект пересекается не более, чем с восемью дочерними октантами на каждом следующем уровне октального дерева, поэтому затраты ограничиваются выражением $8C_I n_i \sum_{j=i+1}^{h} n_j$. Такая оценка априори не более чем в четыре раза превышает вычислительную сложность операции, поскольку в лучшем случае внутренний объект пересекается с двумя дочерними октантами. Таким образом, получаем следующую оценку стоимости локализации столкновений:

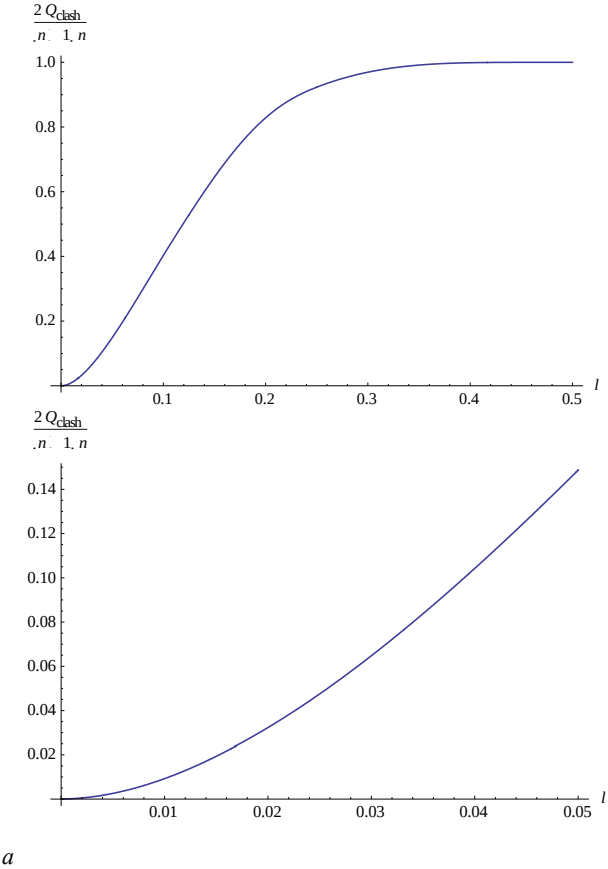
$$Q_{clash} \leq \sum_{i=1}^{h} C_I 8^{i-1} n_i \left(\frac{n_i}{2} + 8 \sum_{j=i+1}^{h} n_j \right)$$

По лемме 2

$$n_i = \begin{cases} n \dfrac{\left(1 - 2^{i-1} l\right)^3 - (1 - 2^i l)^3}{8^{i-1}(1-l)^3}, i = 1..h-1 \\ n \dfrac{(1 - 2^{h-1} l)^3}{8^{h-1}(1-l)^3}, i = h \end{cases}$$

Подставляя n_i и выполнив суммирование рядов, получим требуемое выражение для общей оценки. Как и следовало ожидать, сложность локализации столкновений в общем случае пропорциональна квадрату числа объектов. Особый случай составляют точечные объекты ($l = 0$), для которых полученное выражение принимает вид $\frac{C_I n^2}{2 \cdot 8^{h-1}}$. По построению регулярного октального дерева для точечных объектов выполняется соотношение $\frac{n}{8^{h-1}} \leq m$ и, следовательно, имеет место оценка $Q_{clash} \leq \frac{C_I n m}{2}$. Что и требовалось доказать.

Важным следствием леммы является линейная сложность определения совпадений точечных данных с использованием регулярного октального дерева. Оценим эффект применения октального дерева для поиска столкновений протяженных объектов. На рисунке 5 представлены графики функции стоимости определения столкновений в модельном наборе данных в зависимости от относительных габаритов объектов. Графики построены на основе общей оценки леммы 4, а для наглядности значения стоимости отнесены к пессимистической оценке наивного поиска столкновений $\frac{1}{2} C_I n(n-1)$.



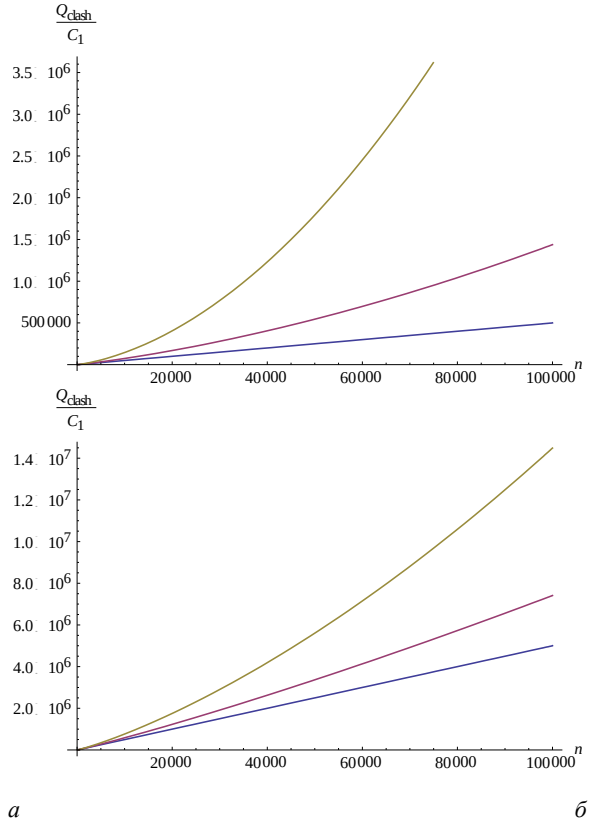
а *б*

Рисунок 5. Графики приведенной стоимости определения столкновений в модельном наборе данных в зависимости от габаритов объектов а) характер функции стоимости на всем диапазоне изменения габаритов объектов б) характер функции стоимости при относительно небольших габаритах объектов.

На графиках видно, что при больших габаритах объектов функция стоимости асимптотически стремится к пессимистической оценке. Для относительно небольших объектов стоимость поиска столкновений существенно ниже, однако быстро нарастает по мере увеличения габаритов объектов и пространственного заполнения сцены.

Проанализируем качественно, как стоимость поиска столкновений зависит от числа объектов и их размеров. На рисунках 6а и 6б приведены графики, иллюстрирующие характер зависимости при разных значениях порога наполнения октантов $m = 10$ и $m = 100$ и вариации габаритов объектов $l = 0$, $l = 0.001$ и $l = 0.003$. Видно, что сложность возрастает с увеличением числа объектов и их размеров. Нижние прямые на графиках соответствуют

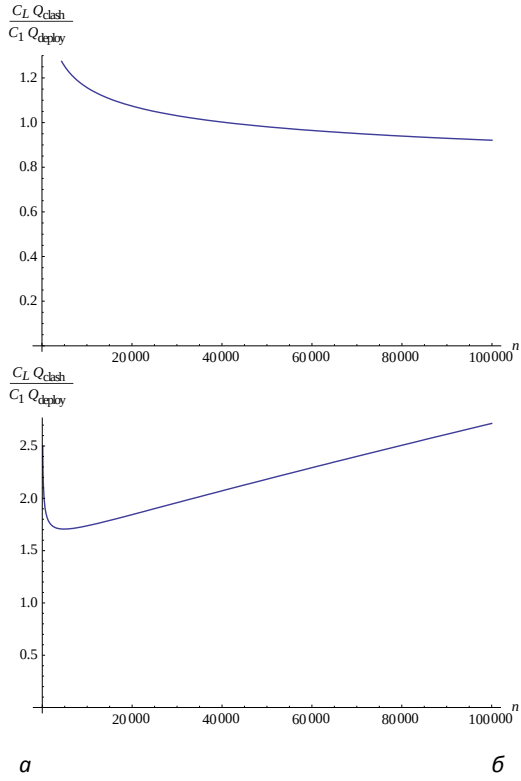
точечным данным, а средние и верхние кривые — протяженным объектам соответствующих размеров. Хотя анализ совпадений точечных данных имеет линейную сложность, а поиск столкновений протяженных объектов — квадратичную, для разреженных сцен различия оказываются не столь существенным даже для значительного числа объектов, составляющего на приведенных графиках 100000. Это обстоятельство оказывается чрезвычайно важным для применения обсуждаемых алгоритмов в индустриальных приложениях, оперирующими большими данными и подверженным быстрой деградации производительности даже при использовании алгоритмов невысокой полиномиальной сложности. Согласно приведенным графикам параметр наполнения октантов также влияет на стоимость поиска столкновений, однако принципиально не меняет характер исследуемых зависимостей.



а *б*

Рисунок 6. График стоимости локализации столкновений в зависимости от количества объектов при различных значениях габаритов ($l = 0,\ l = 0.001,\ l = 0.003$) а) при пороге наполнения октантов $m = 10$ б) при пороге наполнения октантов $m = 100$.

Таким образом, эффект использования регулярных октальных деревьев для быстрого поиска столкновений является значительным. Естественным является вопрос, в какой степени окупаются затраты на развертывание самих деревьев. Для этого построим семейство графиков для функции относительной сложности определения столкновений, приведенной к затратам на развертывание дерева. Построенные графики представлены на рисунке 7 и соответствуют значениям габаритов объектов $l = 0$, $l = 0.001$, $l = 0.01$ и $l = 0.03$ при значении порога наполнения октантов $m = 10$. На графике 7а, соответствующем точечным данным, видно, что затраты на развертывание октального дерева приблизительно соотносятся с затратами на анализ совпадений, а с учетом достигнутого эффекта в производительности всегда окупятся. Различия еще более ощутимы для сцен с протяженными объектами. Затраты на построение дерева становятся пренебрежительно малы по отношению к затратам на определение столкновений. Однако следует иметь в виду, что и эффект от использования октальных деревьев существенно падает с заполнением сцены и необходимостью применения наивного алгоритма для пересечения всех пар объектов в октантах верхних уровней.



а *б*

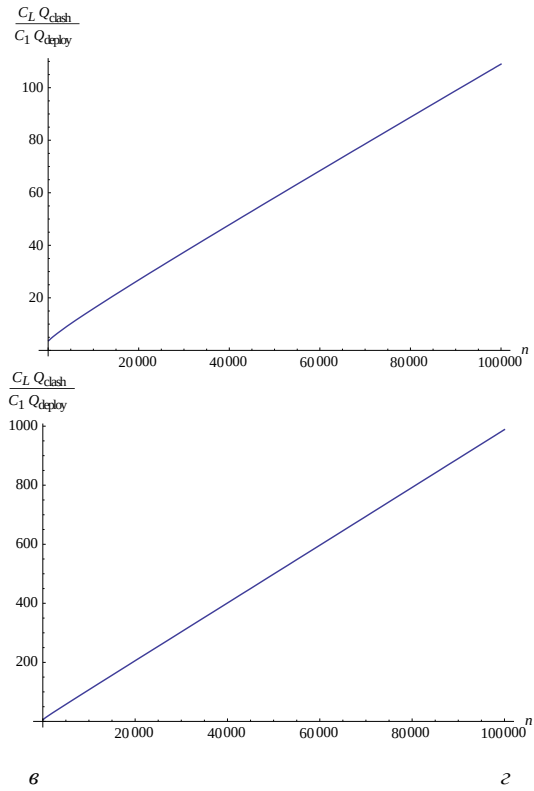
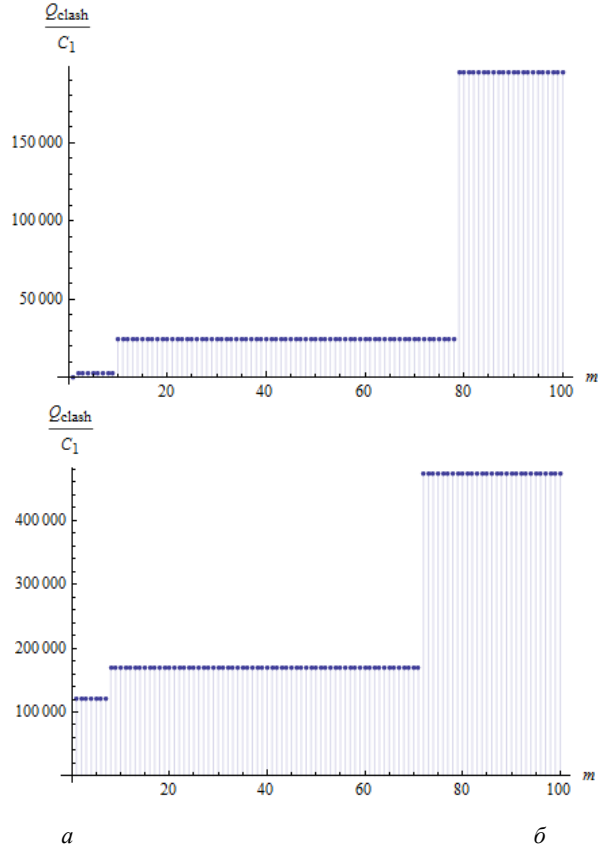


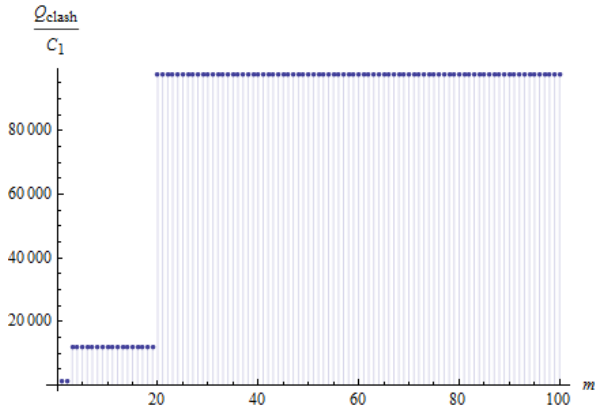
Рисунок 7. График стоимости определения столкновений относительно затрат на развертывание регулярно октального дерева при различных габаритах объектов *а)* $l = 0$ *б)* $l = 0.001$ *в)* $l = 0.01$ *г)* $l = 0.03$

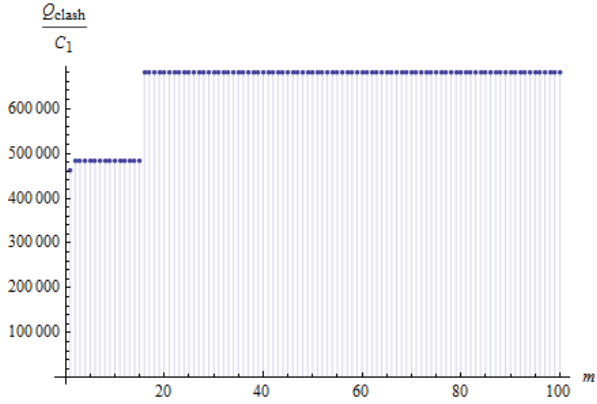
Важным представляется вопрос о выборе порогового значения наполнения октантов m, обеспечивающего минимальные затраты на определение столкновений. Уменьшение порога приводит к неизбежному перераспределению объектов в листовых октантах следующего уровня дерева. С одной стороны, это приводит к более высокой фрагментации объектов и уменьшению затрат на определение столкновений внутри отдельных листовых октантов. С другой стороны, с увеличением глубины дерева возрастают расходы на анализ столкновений объектов, локализованных на верхних уровнях дерева, с объектами, перераспределенными в листьях. Однако проведенный анализ показывает, что второй фактор не является доминирующим и выбор минимального порога всегда приводит к уменьшению затрат на определение столкновений. На рисунке 8 представлено семейство графиков для функции затрат в зависимости от порога наполнения m. Графики построены для разных сочетаний числа объектов и их

относительных габаритов. Как следует из анализа графиков, выбор $m = 10$ является вполне оправданным для всех проанализированных вариантов. Данное значение всегда обеспечивает низкие затраты на определение столкновений. В тоже время, он исключает излишне детальную декомпозицию пространства и неизбежные расходы по памяти на хранение более глубокого представления октального дерева. Действительно, с каждым новым уровнем дерева количество октантов увеличивается в восемь раз, что приводит к быстрому исчерпанию объема доступной памяти приложениями.



а *б*


$\frac{Q_{clash}}{C_1}$
80 000
60 000
40 000
20 000
20
40
60
80
100
m


$\frac{Q_{clash}}{C_1}$
600 000
500 000
400 000
300 000
200 000
100 000
20
40
60
80
100
m

в *г*

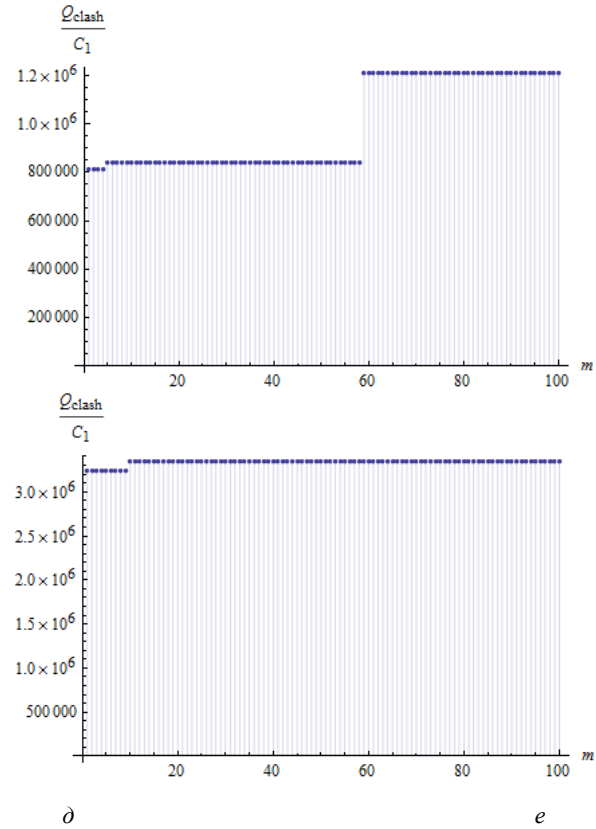


Рисунок 8. График функции затрат на определение столкновений в зависимости от порога наполнения октантов при различных параметрах модельной сцены а) $n = 5000, l = 0$ *б)* $n = 5000, l = 0.01$ *в)* $n = 10000, l = 0$ *г)* $n = 10000, l = 0.01$ *д)* $n = 5000, l = 0.03$ *е)* $n = 10000, l = 0.03$

3.3 Анализ сложности выборки объектов по заданной области

Перейдем к оценкам затрат на выборку объектов, принадлежащих заданной пространственной области или имеющих непустое пересечение с ней. Для определенности будем считать, что область представляет собой куб с относительным размером ребер L, приведенным к габаритам ограничивающего параллелепипеда всего набора данных. Ниже приведен псевдокод алгоритма выборки.

PROCEDURE CLIP (octant, box, result)
POINTER NODE octant
BOX box

```
POINTER OBJECT COLLECTION R

IF(!IS_INTERSECTING(octant,box))
        RETURN

FOR_EACH(OBJECT o IN OBJECTS(octant))
        {
        IF(IS_INTERSECTING(o, box))
                ADD (o, result)
        }

FOR_EACH(NODE child IN CHILDREN(octant))
        CLIP(child, box, result)
```

Алгоритм 3. Выборка объектов по заданной области на основе регулярного октального дерева.

В теле процедуры CLIP октальное дерево обходится с самого верхнего уровня и октанты подвергаются рекурсивной проверке на пересечение с областью выборки. В случае положительного вердикта аналогичной проверке подвергаются все объекты, приписанные к текущему октанту, а также все дочерние октанты, в случае отрицательного вердикта — поиск в глубину текущего октанта прекращается. Объекты, приписанные октантам с ненулевым пересечением области выборки, группируются и формируют окончательный результат.

Определим характерный уровень локализации области выборки как $H = \left\lceil \log_2 \frac{1}{L} \right\rceil$.

Лемма 5. Пусть регулярное октальное дерево глубиной h развернуто для модельной сцены $S(n, l)$. Тогда выборка объектов по области с уровнем локализации H может быть проведена в среднем за Q_{clip} в соответствии со следующей оценкой:

$$Q_{clip}$$

$$\leq \begin{cases} C_I n \dfrac{64 - 224 \cdot 2^h l + 336 \cdot 4^h l^2 + 8^h (11l - 81l^2 + (113 - 56h) l^3)}{8^h (1-l)^3} \\ +8C_I (h-1), h \leq H \\ C_I n \dfrac{64 - 224 \cdot 2^H l + 336 \cdot 4^H l^2 + 8^H (11l - 81l^2 + (113 - 56H) l^3)}{8^H (1-l)^3} \\ +C_I \dfrac{8^{h-H+2} + 56H - 120}{7}, h > H \end{cases}$$

Доказательство:

В соответствии с приведенным алгоритмом вначале определяются октанты, принадлежащие области выборки, а затем выбираются объекты, имеющие ненулевое пересечение с ней. Заметим, что на первом уровне необходимо проанализировать лишь один октант и все приписанные ему объекты. На каждом следующем уровне $i = 2 \dots H$ вплоть до уровня локализации области анализу подлежит не менее одного и не более восьми октантов. На оставшихся уровнях $i = H + 1 \dots h$ достаточно проанализировать не более 8^{i-H+1} октантов. С учетом математического ожидания числа объектов n_i на каждом i-уровне дерева вычислительная сложность выборки может быть оценена как

$$Q_{clip} \leq \begin{cases} C_I\left(n_1 + 8\sum_{i=2}^{H}(n_i+1) + \sum_{i=H+1}^{h} 8^{i-H+1}(n_i+1)\right), h > H \\ C_I\left(n_1 + 8\sum_{i=2}^{h}(n_i+1)\right), h \leq H \end{cases}$$

Отметим, что такая оценка не более чем в 8 раз превосходит минимальные затраты на выполнение выборки. Используя результаты леммы 2 для математического ожидания числа объектов n_i и проводя суммирование рядов, имеем при $h \leq H$

$$Q_{clip}$$

$$\leq C_I n \frac{64 - 224 \cdot 2^h l + 336 \cdot 4^h l^2 + 8^h(11l - 81l^2 + (113 - 56h)l^3)}{8^h(1-l)^3}$$

$$+ 8C_I(h-1)$$

В противоположном случае $h > H$ имеем оценку

$$Q_{clip}$$

$$\leq C_I n \frac{64 - 224 \cdot 2^H l + 336 \cdot 4^H l^2 + 8^H(11l - 81l^2 + (113 - 56H)l^3)}{8^H(1-l)^3}$$

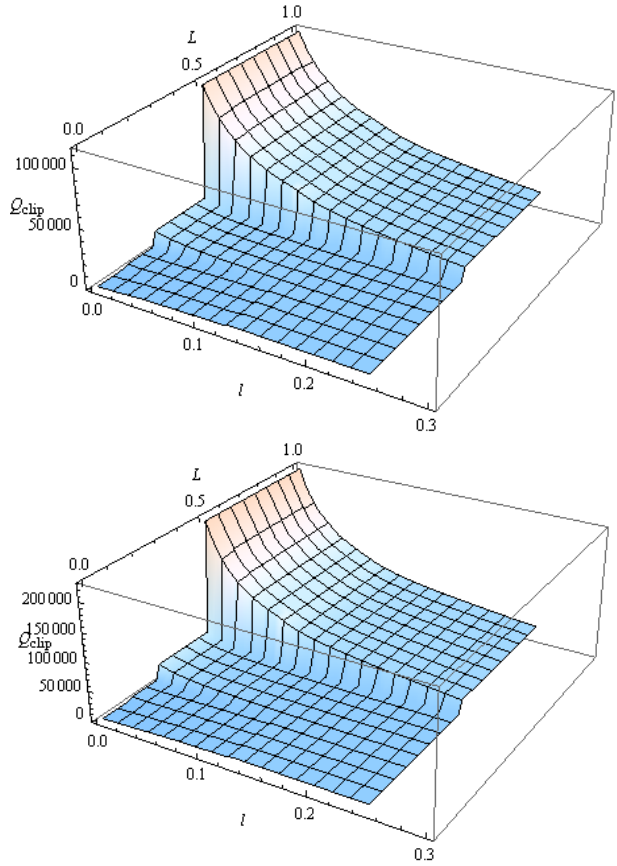
$$+ C_I \frac{8^{h-H+2} + 56H - 120}{7}$$

Что и требовалось доказать.

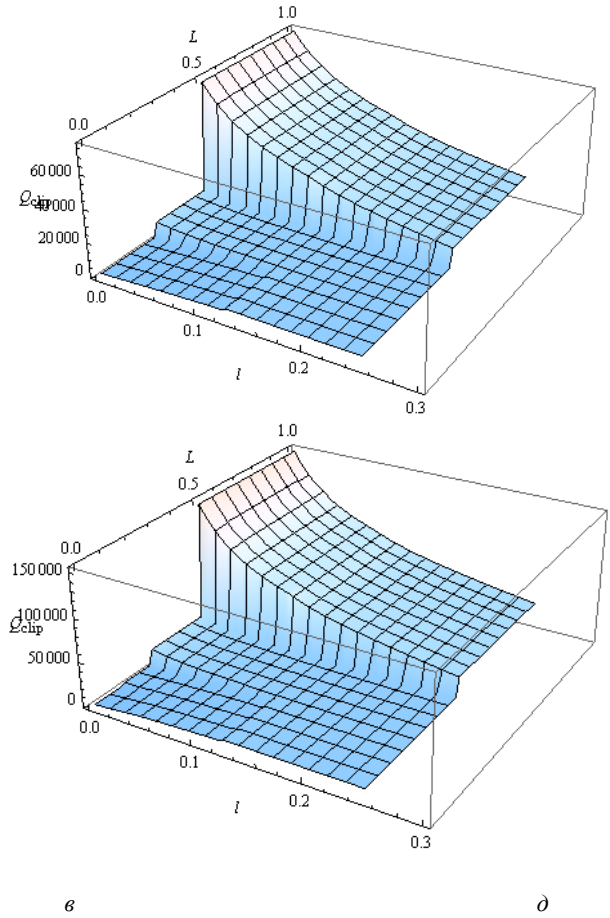
Заметим, что при $H = h$ полученные выражения полностью совпадают. Первые слагаемые в обеих оценках выражают затраты на поиск объектов, принадлежащих области выборки или пересекающие ее, вторые — затраты на

предварительную локализацию октантов, в которых такие объекты могут быть найдены. Полученные результаты согласуются с ранее полученными оценками для точечных данных [9].

На рисунке 9 представлены семейства графиков, иллюстрирующие зависимость стоимости выборки объектов от габаритов объектов и размера области выборки. Как видно, стоимость выборки растет с увеличением размера области выборки и при $L > \frac{1}{4}$ использование октального дерева теряет всякий смысл. При меньших размерах применение октального дерева вполне оправданно, однако рост габаритов объектов может нивелировать суммарный эффект.



а *б*



в *д*

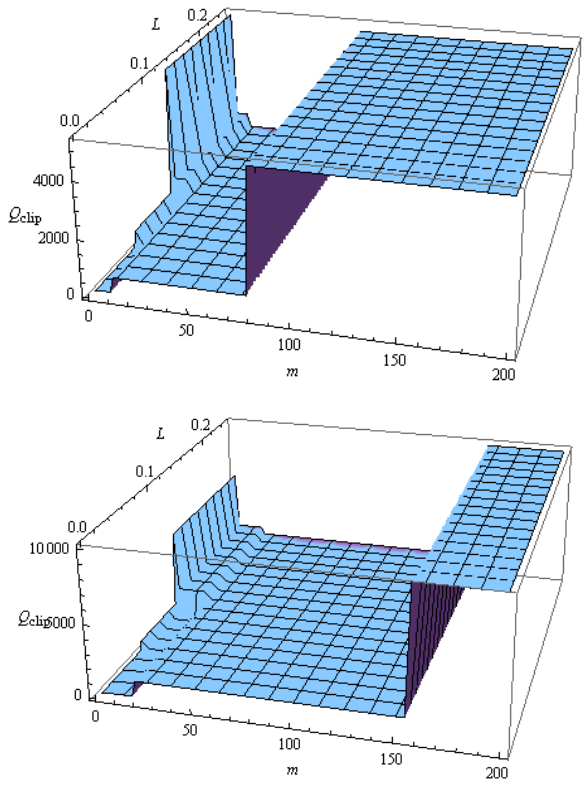
Рисунок 9. Графики стоимости выборки от габаритов объектов и размера области выборки а) $n = 5000, m = 5$ *б)* $n = 10000, m = 5$ *в)* $n = 5000, m = 10$ *г)* $n = 10000, m = 10$

Обсудим вопрос о разумном выборе порога наполнения октантов. Для этого построим соответствующие графики зависимости стоимости выборки от порога наполнения и размеров области выборки. Семейства графиков на рисунке 10 воспроизводят эти зависимости при разных параметрах модельной сцены, а именно: при вариации числа объектов и их размеров. Как видно из этих графиков, затраты на выполнение выборки возрастают с ростом габаритов области выборки. Вместе с тем, зависимость от порога наполнения октантов носит более сложный характер. Так, при больших габаритах области выборки ($L \geq \frac{1}{16}$) выгодны большие значения порога заполнения, поскольку в

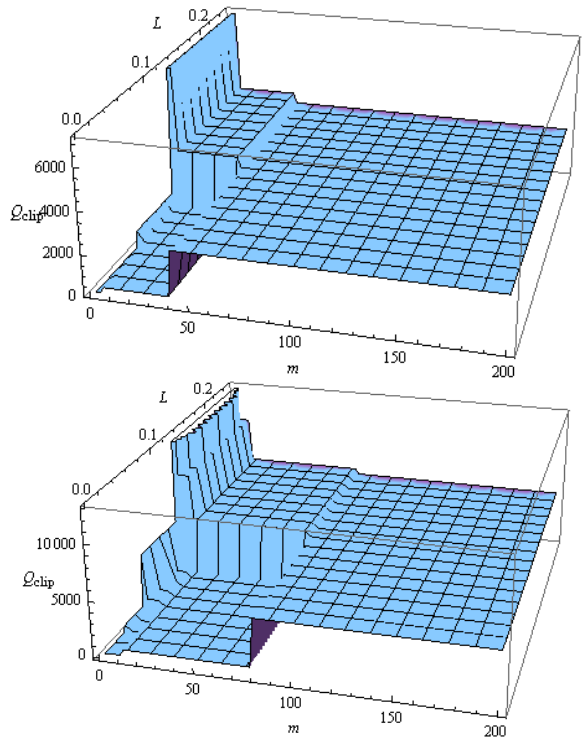
этом случае в стоимости выборки превалирует компонент, связанный с предварительной локализацией октантов, в которых могут содержаться интересующие нас объекты. С другой стороны, при малых габаритах области выборки ($L < \frac{1}{16}$), наибольшее влияние оказывает слагаемое соответствующее затратам на поиск объектов, которые, в свою очередь, пропорциональны значению m.

В исследуемом диапазоне изменения параметров сцены выбор $m = 50$ является вполне приемлемым, поскольку минимизирует затраты на исполнение типовых запросов выборки и при этом не приводит к избыточному расходованию памяти на развертывание глубоких октальных деревьев.

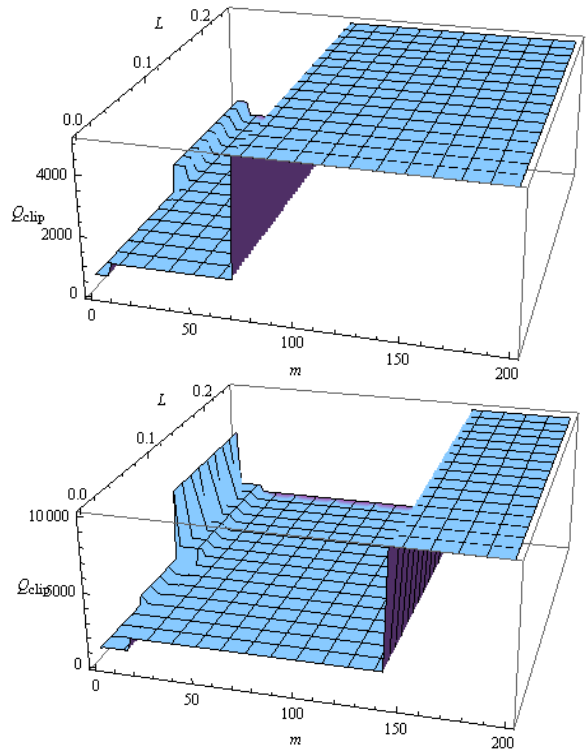
Тем не менее, выбор порога наполнения в общем случае является нетривиальным. Во-первых, рассмотренный модельный набор данных лишь эмулирует упрощенную постановку задачи, связанной с моделированием пространственно-трехмерных сцен. В реальных приложениях сцены могут содержать объекты, которые имеют существенно различные габариты и неравномерно распределены по пространству. Полученные строгие оценки применимы лишь к модельному набору данных, однако правомерность их использования для произвольных сцен, видимо, должна подкрепляться вычислительными экспериментами. Тем не менее, по мере приближения к модельному случаю, мы можем рассчитывать и на адекватность полученных теоретических результатов. Во-вторых, неопределенность с выбором порога наполнения усугубляется тем обстоятельством, что приложения могут консолидировать многие функции одновременно, а их эффективная реализация может предполагать использование совершенно разных значений порога, как в рассмотренных задачах локализации столкновений и выборки по области. В этих случаях, видимо, следует руководствоваться частотой применения отдельных функций и выбора порога, обеспечивающего их равномерно высокую эффективность.


L
0.2
0.1
0.0
4000
Q_{clip}
2000
0
0
50
100
150
200
m
L
0.2
0.1
0.0
10 000
Q_{clip}
5000
0
0
50
100
150
200
m

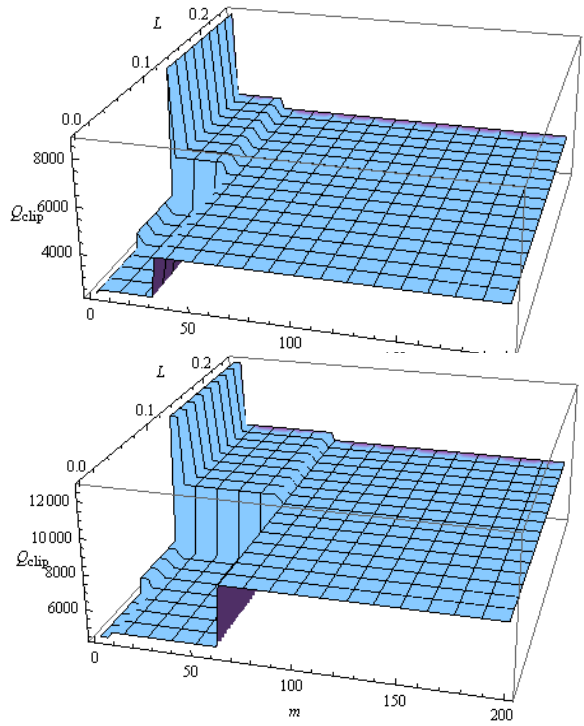
а *б*


L
0.2
0.1
0.0
6000
Q_{clip} 4000
2000
0
0
50
100
150
200
m
L
0.2
0.1
0.0
10 000
Q_{clip}
5000
0
0
50
100
150
200
m

в *г*


L
0.2
0.1
0.0
4000
Qclip
2000
0
0
50
100
150
200
m
L
0.2
0.1
0.0
10 000
Qclip
5000
0
0
50
100
150
200
m

д *е*



ж *з*

Рисунок 10. Графики зависимости стоимости выборки от размеров области и порога наполнения октантов при разных параметрах модельной сцены. а) $n = 5000, l = 0$ *б)* $n = 10000, l = 0$ *в)* $n = 20000, l = 0$ *г)* $n = 40000, l = 0$ *д)* $n = 5000, l = 0.01$ *е)* $n = 10000, l = 0.01$ *ж)* $n = 20000, l = 0.01$ *з)* $n = 40000, l = 0.01$

4. Заключение

Таким образом, рассмотрено важное семейство методов декомпозиции, основанное на регулярных октальных деревьях и обеспечивающее эффективную индексацию, быстрый поиск и анализ больших пространственных данных. Для алгоритмов определения столкновений и выборки объектов по заданной области получены оценки сложности в среднем, которые улучшают известные пессимистические результаты и служат теоретическим обоснованием их применения к более широкому классу приложений. В частности, обсуждены возможности применения рассмотренных алгоритмов к приложениям моделирования сложных пространственно-трехмерных сцен с объектами, имеющими протяженные границы. Существенной стороной проведенного исследования является

попытка определить границы применимости методов декомпозиции, а также выработать практические рекомендации по использованию конкретных алгоритмов в широком диапазоне изменения характеристик пространственных данных.

Список литературы

[1]. M. Cai., P. Revesz, Parametric rectangles: an index structure for moving objects, в *Proceedings of the 10th COMAD International conference on management of data*, 2000.

[2]. V. Semenov, K. Kazakov, S. Morozov, O. Tarlapan, V. Zolotov и T. Dengenis, 4D modeling of large industrial projects using spatio-temporal decomposition, в *eWork and eBusiness in Architecture, Engineering and Construction*, London, UK, pp. 89-95, 2010.

[3]. В. Золотов и В. Семенов, Современные методы поиска и индексации многомерных данных в приложениях моделирования больших динамических сцен, *Труды Института системного программирования,* (24), pp. 381-416, 2013.

[4]. G. Kedem, The quad-CIF tree: a data structure for hierarchical on-line algorithms, в *Proceedings of the 19th Design Automation Conference*, pp. 352-357, 1992.

[5]. B. J. Finkel R.A., Quad trees: a data structure for retrieval on composite keys, *Acta Informatica*, 1(4), pp. 1-9, 1974.

[6]. V. Semenov, K. Kazakov, V. Zolotov, H. Jones и S. Jones, Combined strategy for efficient collision detection in 4D planning applications, в *Computing in Civil and Building Engineering*, Nottingham, UK, pp. 31-39, 2010.

[7]. S. Gottschalk, M. C. Lin и D. Manocha, OBB Tree: a hierarchical structure for rapid interference detection, в *Proceedings of the SIGGRAPH'96 Conference*, New Orleans, USA, pp. 171-180, 1996.

[8]. S. Gottschalk, Collision queries using oriented bounding boxes, Chapel Hill: The University of North Carolina, 2000.

[9]. H. Samet, Foundations of Multidimentional and Metric Data Structures, San Francisco: Morgan Kaufmann, 2006.

On application of spatial decomposition method for large data sets indexing.

Zolotov V.A,. Semenov V.A. (ISP RAS, Moscow, Russia)

Annotation. The paper is dedicated to theoretical research of spatial indexing methods in conformity to three dimensional scenes arising in CAD/CAM systems, robotics, virtual and augmented reality applications. Special attention is given to the decomposition methods based on regular dynamic octrees. A particular version of the octrees is described and investigated to satisfy to the efficiency requirements for typical spatial queries in complex dynamic scenes with the extended borders objects. To perform the needed complexity analysis, the developed octree structure is analyzed against typical spatial queries like collision detection, frustum culling. To obtain more relevant estimates of the complexity on average rather than known pessimistic estimates in worst case, model scene datasets have been introduced. They assume uniform distribution of equal size objects over the scene volume and, being parametric, allow simulation of wide range of industry meaningful scenes. Some auxiliary lemmas about octree depth and distribution of the scene objects over octree levels have been proven based on probabilistic analysis. Final theorem statements provides for the complexity estimates of query evaluation for the model scenes. The obtained results show that asymptotic complexity grows with the object sizes and the scene occupancy. However, the estimates on average improve known results in worst case and can be considered as a theoretical background to introduce the investigated indexing structures to practical applications.

Keywords: spatial indexing, octrees, computational complexity

References

[1]. Cai M., Revesz P. Parametric rectangles: an index structure for moving objects. 2000. Proceedings of the 10th COMAD International conference on management of data.

[2]. Semenov V.A., Kazakov K.A., Morozov S.V., Tarlapan O.A., Zolotov V.A., Dengenis T. 4D modeling of large industrial projects using spatio-temporal decomposition. [ed.] K. Menzel и R. Scherer. London, UK : CRC Press, Taylor & Francis Group, 2010. eWork and eBusiness in Architecture, Engineering and Construction. pp. 89-95.

[3]. Zolotov V.A., Semenov V.A. Sovremennye metody poiska i indeksatsii mnogomernykh dannykh v prilozheniyakh modelirovaniya bol'shikh dinamicheskikh stsen [Advanced indexing methods for large multidimentional data in complex dynamic scenes]. Trudy ISP RAN [The Proceedings of ISP RAS], 2013, vol. 24, pp. 381-416. (in Russian)

[4]. Kedem G. The quad-CIF tree: a data structure for hierarchical on-line algorithms. 1992. Proceedings of the 19th Design Automation Conference. pp. 352-357.

[5]. Finkel R.A., Bentley J.L. Quad trees: a data structure for retrieval on composite keys. Acta Informatica. 1974, vol. 4, 1, pp. 1-9.

[6]. Semenov V.A., Kazakov K.A., Zolotov V.A., Jones H., Jones S. Combined strategy for efficient collision detection in 4D planning applications. [ed.] W. Tizani. Nottingham, UK : Nottingham University Press, 2010. Computing in Civil and Building Engineering. pp. 31-39. ISBN 978-1-907284-60-1.

[7]. Gottschalk S., Lin M. C., Manocha D. OBB Tree: a hierarchical structure for rapid interference detection. New Orleans, USA, 1996. Proceedings of the SIGGRAPH'96 Conference. pp. 171-180.
[8]. Gottschalk S. Collision queries using oriented bounding boxes. Chapel Hill : The University of North Carolina, 2000.
[9]. Samet H. Foundations of Multidimentional and Metric Data Structures. San Francisco : Morgan Kaufmann, 2006.

Автоматическое извлечение новых концептов предметно-специфичных терминов

Д.Г. Федоренко, Н.А. Астраханцев
fedorenko@ispras.ru, astrakhantsev@ispras.ru

Аннотация. В статье описывается способ распознавания предметно-специфичных терминов, которые присутствуют в текущей базе знаний, но выражают отсутствующие в ней концепты. Разработанный метод может быть применен к неформальным базам знаний, поскольку требует только вычисления семантической близости между концептами и статистики встречаемости терминов в корпусе документов. Экспериментальная проверка показывает, что разработанный алгоритм превосходит существующие подходы, а также позволяет повысить точность разрешения лексической многозначности.

Ключевые слова: извлечение концептов; предметно-специфичные термины; обогащение баз знаний; обогащение онтологий; неформальная база знаний; неформальная онтология; разрешение лексической многозначности; семантический анализ.

1. Введение

Лексическая многозначность – неотъемлемая часть естественного языка. Слова и словосочетания могут нести различную смысловую нагрузку в зависимости от контекста, в котором они использовались. В компьютерной лингвистике задача определения значений (смысла) слов на основе контекста называется задачей *разрешения лексической многозначности*. Данная задача в настоящее время является одной из центральных и сложнейших [1] проблем обработки текстов.

Большинство современных методов разрешения лексической многозначности основаны на *базах знаний*, или *онтологиях* [2]. В контексте данной задачи, базой знаний обычно называют совокупность слов, их значений, или *концептов*, и связей между ними. Одна из проблем баз знаний заключается в их неполноте, т.е. в отсутствии подходящих значений слов, которые могут встретиться в некоторых текстах. Данная проблема особенно актуальна для

систем автоматического перевода и информационного поиска, т.к. на вход таких систем обычно подаются произвольные тексты, задаваемые пользователями. Дефицит значений слов отрицательно сказывается на качестве результатов систем, использующих механизм разрешения многозначности [3].

Наиболее часто проблема неполноты проявляется на текстах узких предметных областей, которые слабо покрываются базой знаний. Ключевую часть таких текстов составляют *специфичные термины* – текстовые представления понятий предметной области. Отсутствующие концепты специфичных терминов могут быть добавлены в базу знаний вручную, однако это потребует больших трудозатрат ввиду огромного числа существующих предметных областей.

Таким образом, возникает необходимость автоматического *обогащения базы знаний* – устранения неполноты за счет добавления в базу знаний новых концептов специфичных терминов.

Распознавание новых концептов — один из основных этапов процесса обогащения базы знаний [4]. Данный этап может происходить по-разному, в зависимости от масштаба базы знаний. В случае небольших баз знаний фиксированных предметных областей, обычно считают, что все специфичные термины несут в себе новые концепты [5]. Исключение составляют лишь термины с высоким уровнем специфичности, сам факт наличия которых в базе знаний означает существование для них подходящих концептов (явление однозначности или моносемии [6]). Однако в случае крупных баз знаний, покрывающих различные предметные области, возникает необходимость в более интеллектуальных методах распознавания концептов, поскольку для части специфичных терминов подходящие концепты уже существуют. Например, термин “hand” является специфичным для предметной области “Настольные игры” и обозначает концепт “набор карт, которые в рассматриваемый момент находятся в руках игрока”, однако для этого термина есть и другие концепты, начиная от самого распространенного — “часть тела” — и заканчивая названиями фильмов и фамилиями известных людей.

Также распознавание терминов, выражающих новые концепты, может существенно улучшить результаты алгоритмов разрешения лексической многозначности, поскольку большинство современных методов не позволяют определять случаи, когда выбор концепта невозможен по причине его отсутствия – алгоритм просто возвращает заведомо неверный концепт из доступных в текущей базе знаний [3].

В представленной работе описывается новый подход к распознаванию предметно-специфичных терминов, выражающих новые концепты. Данный подход основан на вычислении статистики встречаемости терминов в коллекции документов и *семантической близости* – функции двух концептов, показывающей, насколько концепты похожи по смыслу между собой. Важной

особенностью разработанного подхода является его применимость к *неформальным базам знаний*, характеризующимся относительно простой структурой, отсутствием формальных аксиом и детальной информации о типах связей между концептами.

Данная статья организована следующим образом. Сначала проводится обзор существующих методов распознавания специфичных терминов, выражающих новые концепты. В разделе 3 описывается разработанный метод. Далее, в разделе 4, представлено описание тестовых сценариев и результаты экспериментов. В заключении приводятся основные результаты работы и направления для дальнейших исследований.

2. Обзор методов распознавания специфичных терминов, выражающих новые концепты

В работе [7] был представлен простейший метод определения терминов, выражающих новые концепты. Данный метод основан на оценке весов концептов термина, присвоенных алгоритмом разрешения лексической многозначности. Так, в случае модели вида

$$P(S_i \mid C_1 \ldots C_n) = \frac{P(S_i)P(C_1 \ldots C_n \mid S_i)}{P(C_1 \ldots C_n)}$$

где S_i — i-ый концепт, C_j — j-ый элемент контекста

весом концепта является число от 0 до 1, показывающее вероятность данного концепта. Предположение метода заключается в том, что если алгоритм возвращает низкий вес концепта, то такой концепт, скорее всего, не подходит и для данного термина на самом деле отсутствует подходящее значение. У данного метода есть два существенных недостатка. Первый заключается в применимости данного метода только для терминов, обладающих более чем одним концептом. Действительно, если у термина известен лишь один концепт, то вероятность выбора этого концепта алгоритмом разрешения многозначности равна единице и такой показатель веса невозможно оценить. Вторым недостатком подхода является то, что, как показали тесты в [3], при высоком пороговом значении веса сильно уменьшается показатель полноты, т.е. многие термины с верными концептами классифицируются как термины с отсутствующими концептами. Таким образом, данный подход представляет лишь теоретический интерес и может быть использован как “нижняя граница” (baseline) в решении поставленной задачи.

Более сложный подход был предложен в [3]. Автор данного подхода рассматривает термины с новыми концептами как “выбросы” из совокупности терминов, концепты которых формируют модель разрешения лексической многозначности. Для определения подобных “выбросов” все термины представляются как точки в n-мерном пространстве, в котором каждая координата является значением определенного признака. Из набора текстов,

называемого обучающей выборкой, извлекаются примеры терминов, для которых концепты известны. После чего на ранее неизвестном наборе текстов, тестовой выборке, алгоритм извлекает примеры терминов и пытается определить, относятся ли они к основному множеству примеров. Если пример достаточно сильно удален от обучающей выборки, то алгоритм полагает, что термин выражает в тексте новый концепт.

В данной статье автор в качестве признаков использует слова из контекста, части речи и именованные сущности. В качестве меры для измерения расстояния между точками используется мера Евклида. Правило, определяющее "выбросы" (новые концепты), заключается в сравнении расстояний от примера до основного множества и от ближайшей к примеру точки до оставшейся части основного множества:

$$p(x) = \frac{d_{xt}}{d_{tt}}$$

Если значение данного отношения больше заданного k (автор полагал k равным единице), то пример является выбросом и выражает новый концепт.

Результаты тестов показали, что данный подход способен находить 72% новых концептов с точностью равной 77%. Однако данный подход рассматривался лишь для одного заранее выбранного термина. Если обобщить его на случай многих терминов, то потребуется огромное количество обучающих данных, что сильно усложнит решение исходной задачи. Следовательно, данный подход также представляет лишь теоретический интерес.

Также исследователями были предложены методы, основанные на структуре и свойствах используемой базы знаний. Так, в работе [8] на основе текстового корпуса и базы знаний вычислялась специальная функция близости, позволяющая определять концепты, которые наименее близки к уже существующим.

В работе [9] задача поиска новых концептов рассматривалась как задача иерархической кластеризации с помощью нейронных сетей, строящихся на основе иерархии концептов существующей базы знаний.

На данный момент существует небольшое количество работ, затрагивающих задачу распознавания новых концептов. Самые простые методы показывают достаточно низкие результаты и поэтому могут быть использованы только в качестве "нижней границы" решения рассматриваемой задачи. Более сложные методы имеют ограниченное применение, т.к. зачастую сильно связаны со свойствами конкретных баз знаний. Обычно данные свойства заключаются в поддержке различных видов отношений между концептами [8] [10] и в возможности вычисления специальных функций близости.

3. Описание алгоритма

На вход алгоритма подается коллекция документов некоторой предметной области, а также предметно-специфичные термины, взятые из данной коллекции. Далее следует этап распознавания, результатом которого является список терминов, выражающих новые концепты.

Мы рассматриваем задачу распознавания специфичных терминов с новыми концептами как задачу классификации на два класса: “термин, выражающий существующий в текущей базе знаний концепт” и “термин, выражающий новый концепт”. Данная задача может быть решена с помощью методов машинного обучения, для применения которых необходимо перейти к признаковому описанию объектов.

3.1 Максимальная семантическая близость к ключевым концептам

Данный признак основан на предположении, что подходящий концепт специфичного термина, если он присутствует в текущей базе знаний, должен быть семантически близок к ключевым концептам предметной области. *Ключевые концепты* – это концепты, которые в совокупности формируют высокоуровневое описание документа или коллекции документов. Например, для предметной области “Настольные игры” ключевыми могут являться такие концепты, как “игровое поле”, “кость”, “карта”.

Для проверки данного предположения, для каждого известного концепта специфичного термина вычисляется семантическая близость к ключевым концептам предметной области, после чего величиной признака становится максимальная полученная близость:

$$relatedness(concepts,keyconcepts) = max_{c \in concepts} \sum similarity(c,k)$$

где *similarity(c, k)* – функция семантической близости между двумя концептами.

Ключевые концепты могут быть заданы вручную, либо определены автоматически с помощью одного из существующих алгоритмов [11] [12].

3.2 Отношение числа концептов к количеству вхождений термина

Специфичные термины, по предположению большинства методов построения баз знаний [5], должны выражать единственный концепт на протяжении всей коллекции документов. Например, слово “стек” в текстах о программировании всегда означает структуру данных и крайне редко может означать, например, трость.

Выполнимость данного предположения может быть проверена путем применения алгоритма разрешения лексической многозначности, основанного на рассматриваемой базе знаний, к коллекции документов и анализа его результатов для каждого вхождения термина. Для этого рассматривается отношение числа уникальных концептов, присвоенных термину, к общему количеству вхождений данного термина.

Чем меньше значение данной величины, тем реже алгоритм разрешения лексической многозначности присваивает термину разные концепты. В общем случае данный признак показывает, как часто меняются контексты термина и насколько сильно смена контекста влияет на выбор итогового концепта термина алгоритмом разрешения лексической многозначности.

3.3 Нормализованное число выбранных концептов

Предыдущий признак показывает, как часто алгоритм разрешения лексической многозначности меняет концепты термина, однако он не учитывает общее количество использованных концептов. Рассмотрим следующую ситуацию: алгоритм разрешения лексической многозначности определил 5 различных концептов термина, в то время как число вхождений термина равно 100. Тогда отношение числа концептов к количеству вхождений будет равно 0.05 и, следовательно, данный термин, вероятно, выражает существующий концепт. Однако данный признак перестает быть информативным, когда общее число концептов термина равно 5. Такие ситуации могут быть определены, если дополнительно рассматривать число выбранных алгоритмом разрешения многозначности концептов, поделенное на общее количество концептов, существующих в базе знаний для данного термина.

3.4 Специфичность термина

Данный признак показывает, насколько термин специфичен по отношению к внешнему корпусу общей тематики. Высокая специфичность может указывать на то, что термин обычно употребляется в единственном значении. Например, такие термины, как “синхрофазотрон” и “рибонуклеиновая кислота” обладают высоким уровнем специфичности и имеют единственное значение вне зависимости от контекста употребления. Таким образом, если специфичный термин уже существует в базе знаний, значит для данного термина существует и подходящий концепт.

В качестве признака специфичности используется *Релевантность домену (Domain Relevance)* [13], равная отношению числа вхождений термина в заданной коллекции документов к числу вхождений в коллекции общей тематики:

$$DR(t) = \frac{TF_{target}(t)}{TF_{target}(t) + TF_{reference}(t)}$$

4. Эксперименты

4.1 Тестовые данные

Для экспериментальной проверки предложенного алгоритма была использована база знаний системы Texterra [14], разрабатываемой в Институте Системного Программирования РАН. Данная база знаний состоит из терминов и концептов, полученных на основе англоязычной версии ресурса Википедия и содержит порядка четырех миллионов концептов различных предметных областей.

Одной из тематик, не полностью покрываемой базой знаний системы Texterra, является тематика “Настольные игры”. С целью проверки разработанного метода на данной предметной области, была собрана коллекция документов, состоящая из 1246 статей о настольных играх на английском языке. Для данной коллекции также был составлен список из 75 специфичных терминов, взятых из глоссария сайта Board Games Geek [1]. Каждый из этих терминов был вручную размечен либо как термин с существующим подходящим концептом, либо как термин, для которого в базе знаний системы Texterra нет подходящих концептов. В результате данной разметки было получено следующее соотношение терминов с новыми и существующими концептами: 32 и 43 соответственно.

Примеры терминов, выражающих новые концепты: movement, cell, tile-laying, chrome, crowning.

Примеры терминов, выражающих существующие концепты: dungeon crawl, rock-paper-scissors, replay value, dexterity game, deck.

4.2 Методика тестирования

В качестве методики тестирования использовался метод перекрестной проверки (*кросс-валидация*) со следующим соотношением тренировочного и тестового множеств: 90% и 10% терминов соответственно.

Для оценки качества алгоритмов вычислялись метрики точности и полноты. *Точность* — отношение числа правильно определенных терминов, выражающих новые концепты, к общему числу терминов, классифицированных алгоритмом как термины с новым концептами. *Полнота*

[1] Адрес глоссария: http://boardgamegeek.com/wiki/page/Glossary

— отношение числа правильно определенных терминов, выражающих новые концепты, к общему числу присутствующих в тестовой выборке терминов с новыми концептами.

При подсчете признаков, описанных в разделе 3, использовались алгоритмы разрешения лексической многозначности [14] и поиска ключевых концептов [11], реализованные в системе Texterra. В качестве корпуса общей тематики при подсчете признака специфичности использовался открытый корпус n-грамм современного английского языка [2].

В качестве алгоритмов машинного обучения использовались следующие методы: наивный Байесовский классификатор [15], случайный лес [16] и логистическая регрессия [17].

4.3 Результаты

4.3.1 Распознавание новых концептов

Целью данного теста является вычисление метрик качества разработанного метода. В табл. 1 представлены результаты тестирования различных алгоритмов машинного обучения, реализующих разработанный метод, и "нижней границы" – алгоритма на основе оценки веса концепта (см. раздел 2), при этом итоговым весом считался максимальный вес среди всех вхождений термина в заданной коллекции документов.

Алгоритм	**Точность**	**Полнота**	**F-мера**	**Аккуратность (accuracy)**
Наивный Байесовский классификатор	71.37%	**85.88%**	77.91%	78.26%
Случайный лес	74.13%	75.05%	74.58%	77.46%
Логистическая регрессия	**74.18%**	83.97%	**78.77%**	**80.16%**
Метод на основе оценки веса значения, threshold=0.7	63.5%	67.93%	65.64%	69.93%

Табл. 1 Результаты распознавания новых концептов.

[2] Корпус доступен по адресу: http://www.ngrams.info

Как видно из таблицы, наилучшая точность на классе терминов с новыми концептами достигается с помощью методов логистической регрессии и случайного леса. В то же время наивный Байесовский классификатор показал наилучшую полноту результатов. По показателю аккуратности (точности по обоим классам) и F-меры (среднего гармонического точности и полноты) наилучшим оказался метод логистической регрессии.

Нужно также отметить, что все алгоритмы показали значительно более высокие результаты, чем метод на основе оценки веса концепта.

4.3.2 Разрешение лексической многозначности специфичных терминов

Целью данного теста является определение того, насколько повышается точность разрешения лексической многозначности за счет распознавания терминов, для которых невозможно выбрать правильные концепты ввиду их отсутствия в базе знаний.

В табл. 2 представлены результаты сравнения алгоритма разрешения многозначности системы Texterra и его комбинации с разработанным методом, при этом считалось, что для термина выбран правильный концепт, если он выбран хотя бы для одного вхождения данного термина.

	Точность	**Полнота**	**F-мера**
Texterra	52%	**52%**	52%
Texterra + разработанный метод	**78%**	44%	**56%**

Табл. 2 Результаты разрешения лексической многозначности специфичных терминов.

Как видно из таблицы, разработанный метод позволяет повысить точность разрешения многозначности специфичных терминов с 52% до 78%.

5. Заключение

В данной работе представлен метод распознавания предметно-специфичных терминов, которые присутствуют в текущей базе знаний, но выражают отсутствующие в ней концепты. Данный метод основан на вычислении статистики встречаемости терминов в корпусе документов и семантической близости между концептами. Важной особенностью разработанного подхода является его применимость к неформальным базам знаний, характеризующимся относительно простой структурой.

Разработанный метод был протестирован на основе базы знаний системы Texterra и превзошел существующие подходы. Также было выявлено, что разработанный метод позволяет существенно повысить точность разрешения лексической многозначности специфичных терминов.

Одним из направлений дальнейшей работы является кросс-доменное тестирование метода, то есть использование модели классификатора, обученного на одной предметной области, для распознавания специфичных терминов, извлеченных из коллекции отличной тематики.

Список литературы

[1]. Mallery J. C. Thinking about foreign policy: Finding an appropriate role for artificially intelligent computers // Master's thesis, MIT Political Science Department. 1988.

[2]. Turdakov D. Y. Word sense disambiguation methods // Programming and Computer Software. 2010. 36. No 6. P. 309–326.

[3]. Erk K. Unknown word sense detection as outlier detection // Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics. 2006. P. 128–135.

[4]. Астраханцев Н., Турдаков Д. Методы автоматического построения и обогащения неформальных онтологий // Программирование. 2013. 39. No 1. С. 23–34.

[5]. Biemann C. Ontology learning from text: A survey of methods // LDV forum. 20. 2005. P. 75–93.

[6]. Agirre E., Edmonds P. G. Word sense disambiguation: Algorithms and applications. Springer Science+ Business Media. 2006. 33.

[7]. Erk K., Pado S. Shalmaneser–a toolchain for shallow semantic parsing // Proceedings of LREC. 6. 2006.

[8]. Faatz A., Steinmetz R. Ontology enrichment with texts from the www // Semantic Web Mining. 2002. P. 20.

[9]. Chifu E. T., Le Ia I. A. Text-based ontology enrichment using hierarchical self-organizing maps. 2008.

[10]. Georgiu M., Groza A. Ontology enrichment using semantic wikis and design patterns.

[11]. Grineva M., Grinev M., Lizorkin D. Effective extraction of thematically grouped key terms from text // Proc. of the AAAI 2009 Spring Symposium on Social Semantic Web. 2009. P. 39–44.

[12]. El-Beltagy S. R., Rafea A. KP-Miner: A keyphrase extraction system for English and Arabic documents //Information Systems. 2009. 34. No 1. P. 132-144.

[13]. Pazienza M. T., Pennacchiotti M., Zanzotto F. M. Terminology extraction: an analysis of linguistic and statistical approaches // Knowledge Mining. Springer. 2005. P. 255–279.

[14]. Ivannikov V., Turdakov D., Nedumov Y. Fast Text Annotation with Linked Data. Eighth International Conference on Computer Science and Information Technologies 26–30 September. 2011. Yerevan, Armenia.

[15]. Manning C. D., Schutze H. Foundations of statistical natural language processing. MIT press. 1999. P. 237.

[16]. Breiman L. Random forests // Machine learning. 2001. 45. No 1. P. 5–32.

[17]. Manning C. D., Schutze H. Foundations of statistical natural language processing. MIT press. 1999. P. 589–594.

Automatic Extraction of New Concepts from Domain-Specific Terms

D.G. Fedorenko, N.A. Astrakhantsev
ISP RAS, Moscow, Russia
fedorenko@ispras.ru, astrakhantsev@ispras.ru

Abstract. Most of the state-of-the-art approaches for word sense disambiguation (WSD) are based on knowledge bases, or ontologies — databases of terms, their concepts and relations between them. One of the standing problems of knowledge bases is their incompleteness, i.e. the lack of appropriate concepts for terms occurred in some contexts; the problem is mostly actual for domain-specific terms. The consequence is that systems produce incorrect results because existing WSD algorithms simply assign one of the a-priori incorrect concepts to the terms.

This paper describes a novel approach for recognition of domain-specific terms that exist in the knowledge base but represent new concepts. In contrast to previous approaches requiring formal ontologies with hierarchical structure and different relation types, our method can be applied to informal knowledge bases — it requires only semantic similarity between concepts and statistics of terms extracted from the domain-specific corpus.

We show that our method performs better than existing approaches and achieves 74% precision and 83% recall for the collection of domain-specific terms not fully covered by our knowledge base. Also our method improves precision of WSD from 52% to 78% for the considered terms.

Keywords: concept extraction; domain-specific terms; knowledge base enrichment; ontology enrichment; informal knowledge base; informal ontology; word sense disambiguation; semantic analysis.

References

[1]. Mallery J. C. Thinking about foreign policy: Finding an appropriate role for artificially intelligent computers. Master's thesis. MIT Political Science Department, 1988.
[2]. Turdakov D. Y. Word sense disambiguation methods. Programming and Computer Software, 2010. vol. 36. no 6. pp. 309–326. doi: 10.1134/S0361768810060010
[3]. Erk K. Unknown word sense detection as outlier detection. Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics, 2006. pp. 128–135. doi: 10.3115/1220835.1220852
[4]. Astrakhantsev N., Turdakov D. Automatic construction and enrichment of informal ontologies: A survey. Programming and Computer Software, 2013. vol. 39. no 1. pp. 23–34. doi: 10.1134/S0361768813010039
[5]. Biemann C. Ontology learning from text: A survey of methods. LDV forum. 20. 2005. pp. 75–93.
[6]. Agirre E., Edmonds P. G. Word sense disambiguation: Algorithms and applications. Springer Science+ Business Media. 2006. 33.
[7]. Erk K., Pado S. Shalmaneser–a toolchain for shallow semantic parsing. Proceedings of LREC. 6. 2006.

[8]. Faatz A., Steinmetz R. Ontology enrichment with texts from the www. Semantic Web Mining, 2002.
[9]. Chifu E. T., Le Ia I. A. Text-based ontology enrichment using hierarchical self-organizing maps. 2008.
[10]. Georgiu M., Groza A. Ontology enrichment using semantic wikis and design patterns.
[11]. Grineva M., Grinev M., Lizorkin D. Effective extraction of thematically grouped key terms from text. Proc. of the AAAI 2009 Spring Symposium on Social Semantic Web, 2009. pp. 39–44.
[12]. El-Beltagy S. R., Rafea A. KP-Miner: A keyphrase extraction system for English and Arabic documents. Information Systems, 2009. vol. 34, no 1. pp. 132-144. doi: 10.1016/j.is.2008.05.002
[13]. Pazienza M. T., Pennacchiotti M., Zanzotto F. M. Terminology extraction: an analysis of linguistic and statistical approaches. Knowledge Mining. Springer, 2005. pp. 255–279. doi: 10.1007/3-540-32394-5_20
[14]. Ivannikov V., Turdakov D., Nedumov Y. Fast Text Annotation with Linked Data. Eighth International Conference on Computer Science and Information Technologies 26–30 September. 2011. Yerevan, Armenia.
[15]. Manning C. D., Schutze H. Foundations of statistical natural language processing. MIT press, 1999. p. 237.
[16]. Breiman L. Random forests. Machine learning, 2001. vol. 45, no 1. pp. 5–32. doi: 10.1023/A:1010933404324
[17]. Manning C. D., Schutze H. Foundations of statistical natural language processing. MIT press, 1999. pp. 589–594.

Определение демографических атрибутов пользователей микроблогов[1]

Антон Коршунов, Иван Белобородов, Андрей Гомзин, Кристина Чуприна, Никита Астраханцев, Ярослав Недумов, Денис Турдаков
{korshunov, ivbel, gomzin, chuprina, astrakhantsev, yaroslav.nedumov, turdakov}@ispras.ru

Аннотация. При заполнении полей профиля в различных интернет-сервисах пользователи зачастую по ошибке или преднамеренно не указывают значения некоторых демографических атрибутов, таких как пол, возраст, семейное положение, уровень образования, религиозные и политические взгляды. Вместе с тем, информация об атрибутах пользователей позволяет существенно повысить эффективность систем рекомендации, интернет-маркетинга и других приложений, предполагающих персонализацию результатов. В статье предлагается метод автоматического определения демографических атрибутов пользователей социального сервиса микроблогов Twitter по текстам их сообщений и другой доступной информации из профилей. Метод основан на алгоритме машинного обучения, его отличительными особенностями являются полностью автоматическое построение исходного набора данных для обучения и тестирования, а также поддержка широкого набора языков и демографических атрибутов. Экспериментальные исследования показали высокое качество результатов определения пола, возраста и семейного положения пользователя для наиболее популярных языков: английского, русского, немецкого, французского, итальянского и испанского. Кроме того, для английского языка поддерживается также определение уровня образования, а также религиозных и политических взглядов пользователя.

Ключевые слова: демографические характеристики; демографические атрибуты; социальные сети; микроблоги; обработка текстов на естественном языке; анализ содержимого; компьютерная лингвистика; машинное обучение.

1. Введение

В связи с увеличением количества пользователей интернета, а также появлением новых средств для обмена информацией, количество свободно

[1] Работа выполнена при финансовой поддержке Минобрнауки Российской Федерации по государственному контракту от 10.10.2013 г. № 14.514.11.4111 в рамках ФЦП «Исследования и разработки по приоритетным направлениям развития научно-технологического комплекса России на 2007-2013 годы»

доступных персональных данных (включая текстовые сообщения) постоянно растёт. Учитывая склонность пользователей интернета к анонимности, актуальны методы частичной идентификации авторов сообщений по значениям их демографических атрибутов. В частности, в системах интернет-маркетинга и рекомендаций особую важность представляет определение демографических атрибутов пользователя для таргетированного продвижения товаров и услуг в группах пользователей с одинаковыми значениями атрибутов. Помимо интернет-сервисов, такие социо-демографические характеристики находят применение в различных дисциплинах: социология, психология, криминология, экономика, управление персоналом и др.

Демографические атрибуты можно условно разделить на категориальные (пол, национальность, раса, семейное положение, уровень образования, профессия, трудоустроенность, религиозные и политические взгляды) и численные (возраст, уровень доходов). Условность разделения связана с тем, что значения численного атрибута можно отобразить в набор категорий и в дальнейшем рассматривать этот атрибут как категориальный. В частности, значения возраста можно разделить на несколько возрастных категорий, что часто применяется на практике. Набор атрибутов одного пользователя составляет его социо-демографический профиль.

В статье предложен метод определения демографических атрибутов пользователей сети Twitter по текстам их сообщений, обладающий следующими преимуществами:

1. широкий набор поддерживаемых атрибутов: пол, возраст, семейное положение, уровень образования, религиозные и политические взгляды;
2. полностью автоматический метод сбора и разметки корпусов сообщений пользователей интернета для всех поддерживаемых атрибутов;
3. поддерживаемые языки: русский, английский, испанский, немецкий, французский, итальянский;
4. высокое качество результатов.

Дальнейшее изложение строится следующим образом. Раздел 2 содержит обзор литературы. Раздел 3 посвящён деталям предложенного метода. В разделе 4 описываются полученные экспериментальные результаты.

2. Обзор литературы

Задача определения скрытых демографических атрибутов пользователей интернета по текстам их сообщений сводится к классической задаче социолингвистики: определению характерных особенностей языка представителей различных социальных групп, позволяющих производить частичную идентификацию человека по принадлежности к этим группам

(автороведческая экспертиза). Вместе с тем, существует ряд отличий рассматриваемой задачи, связанных с электронной сообщений пользователей:

- ограниченная длина - вследствие ограничения на длину сообщений и/или стремления пользователей сэкономить время длина сообщений редко превышает нескольких сотен символов;
- неформальный стиль - в сообщениях часто встречаются нестандартные аббревиатуры, слэнг, неологизмы, пользователи сознательно пренебрегают классическими нормами орфографии и пунктуации, что можно расценивать как следствие общей тенденции переноса повседневного неформального общения в социальную сеть;
- использование специфических лингвистических конструкций - пользователи активно используют специальные синтаксические конструкции для быстрой категоризации сообщений (теги), добавления ссылок на других пользователей, описания собственных эмоций (эмотиконы) и т.д.;
- нестабильность качества пользовательского контента (большое количество спама и ложных аккаунтов пользователей) обуславливает необходимость в методах фильтрации зашумленных данных.

В связи с перечисленными особенностями и вследствие ограниченной применимости классических методов атрибуции текста к электронным сообщениям пользователей интернета, в последнее десятилетие было предложено множество методов, специализированных для сервисов мгновенных сообщений, электронной почты, форумов, блогов, социальных сетей и других источников текстовых сообщений пользователей. Эти методы, в основном, основаны на применении методов машинного обучения с учителем с целью классификации пользователей по лингвистическим и другим признакам в предопределённые классы, соответствующие различным значениям изучаемых атрибутов. Сообщения пользователя рассматриваются как набор символьных строк, из которых извлекаются признаки, а для разметки обучающей выборки применяются дополнительные источники данных о пользователе.

2.1 Задача определения гендерной принадлежности

Наиболее простым способом определения пола пользователя некоторого Интернет-ресурса является определение пола пользователя по имени, указанному в его профиле. Этот подход используется, например, в работе [1]. Недостатки подхода очевидны: вместо действительных имен пользователи могут указывать псевдонимы, также существует ряд имен, универсальных для обоих полов.

В современных исследованиях, посвященных задаче определения пола, активно используются методы машинного обучения. С точки зрения методов машинного обучения задачу определения пола интернет-пользователя можно

рассматривать как задачу бинарной классификации, где признаки объектов выбираются из размещаемой пользователями информации.

В исследовании [2] для гендерной классификации пользователей социальной сети Facebook помимо словарей имен использовались признаки, являющиеся значениями полей «interested_in» и «relationship_status» профилей пользователей, также учитывалось гендерное распределение друзей каждого пользователя.

Во многих исследованиях используются признаки, являющиеся N-граммами символов и слов сообщений пользователей. Так, исследования [3], [4] используют N-граммы символов длины от 1 до 5, извлекаемые из текстов сообщений, для определения пола пользователей социальной сети Twitter. Следует отметить, что при использовании N-грамм в качестве признаков необходимо осуществлять выбор наиболее репрезентативных N-грамм во избежание получения пространства признаков огромной размерности. Выбор наиболее репрезентативных N-грамм позволяет не только сократить время работы алгоритма, но и улучшить его производительность [4].

В [5] исследуется улучшение качества классификации пользователей сети Twitter при добавлении к признакам – N-граммам символов и слов, извлекаемым из текстов сообщений пользователей, N-грамм, извлекаемых из метаданных (полей «Screen name», «Full name», «Description» профиля пользователя). В [6] для классификации пользователей сети Facebook на основе текстов их статусов в качестве признаков в дополнение к N-граммам слов использовались темы, извлекаемые из текстов статусов при помощи тематического моделирования.

Также можно выделить группу структурных признаков:

1. признаки на основе символов (общее количество символов, общее количество букв, общее количество букв в верхнем регистре, общее количество цифр, общее количество пробелов и т.д.);
2. признаки на основе слов (общее количество слов, средняя длина слова в символах, общее количество различных слов, общее количество коротких слов – слов, длина которых меньше трех символов, количество слов, длина которых превосходит шесть символов, и т.д.);
3. признаки на основе предложений (общее количество различных символов препинания);
4. признаки на основе всего текста (общее количество предложений, общее количество абзацев, среднее число предложений в абзаце, среднее число слов в абзаце, среднее число слов в предложении и т.д.).

В исследовании [7] структурные признаки 1–3 были использованы для классификации пользователей сервиса Youtube. В исследовании [8] структурные признаки 1–4 использовались для гендерной классификации авторов новостных статей и электронных писем. Также, наряду со структурными признаками, в [8] использовались и социолингвистические признаки. В исследованиях [9] и [10] для гендерной классификации

пользователей социальных сетей наряду с признаками – N-граммами использовались социолингвистические признаки. В [11] социолингвистические признаки использовались для гендерной классификации авторов блогов.

В исследованиях [12], [13] для гендерной классификации пользователей сети Twitter использовались признаки, получаемые на основе первых k слов, стемм, хештегов, диграмм и триграмм, встречающихся в текстах пользователей обоих классов. В [12] также исследовалась влияние использования поля имени пользователя на качество работы алгоритма, в [13] – влияние окрестности пользователя в социальном графе.

2.2 Задача определения возраста

Существует три основных подхода к постановке задачи определения возраста. При первом подходе ставится задача определения возраста как непрерывной переменной.

При втором подходе ставится задача определения возрастной категории. Например, можно поставить задачу определения одной из следующих категорий: до 20 лет, от 20 до 40 лет, 40 лет и больше.

Также существует ряд задач, для которых важен не столько возраст пользователя, сколько жизненный этап, на котором он находится. Примерами жизненных этапов могут являться следующие: учащийся школы, студент университета, пенсионер.

Для решения задачи определения возраста широко используются методы машинного обучения. Так, например, в исследовании [14] задача определения возраста была поставлена как задача классификации с двумя классами: 40-, 40+; в исследовании [9] — как задача классификации с двумя классами: 30-, 30+; в исследованиях [15], [16] — как задача классификации с тремя классами: 13–17, 23–27, 33–42; в исследовании [17] — как регрессионная задача. В [18] регрессионные оценки возраста используются для классификации по возрастным категориям. В исследовании [19] проводится сравнение между тремя перечисленными выше подходами для задачи определения возраста пользователей социальной сети Twitter.

Исследования по определению возраста проводились для разнообразных типов данных: блогов ([15], [16]), записей телефонных раговоров ([14]), данных социальных сетей ([9], [19], [20]). Признаки, используемые для решения задачи определения возраста, можно разделить на две основные группы:

1. N-граммы слов и символов, получаемые из текстов сообщений ;
2. стилистические признаки (такие, как части речи, сленг, средняя длина предложения, пунктуация, акронимы, эмотиконы и т.д.).

Также используются признаки, специфические для источника данных. Например, в работе [21] для определения возраста пользователей платформы

LiveJournal использовались такие признаки, как количество друзей пользователя, количество постов, отображаемых на странице пользователя, общее количество постов пользователя, среднее число комментариев к посту пользователя.

2.3 Определение других атрибутов

В [22] описывается определение политической ориентации и этнической принадлежности пользователей сети Twitter с использованием признаков, извлекаемых из профиля пользователя (длина имени, количество букв в имени, количество цифр в имени и т. д.), особенностей поведения (общее количество сообщений, среднее число сообщений в день, среднее время между сообщениями и т. д.), текстового содержимого сообщений (наличие слов, характерных для классов, между которыми проводится классификация), окружения пользователя.

В [23] проводится сравнение двух подходов к определению политической ориентации пользователей сети Twitter. Первый из рассмотренных подходов основывался на классификации пользователей при помощи алгоритма машинного обучения «Метод опорных векторов», использовавшего признаки, извлекаемые из текстов сообщений пользователей; второй подход использовал алгоритм обнаружения сообществ.

В [24] для решения задачи определения географического положения пользователя социальной сети Twitter применяются методы тематического моделирования. В [25] описывается система, определяющая географическое положение пользователя сети Twitter на основе распределения слов по географическим локациям.

2.4 Выводы

Проведен обзор научного направления, связанного с проблематикой задачи, рассматриваемой в рамках данной НИР. Рассмотрены существующие подходы к решению задачи определения таких социо-демографических атрибутов пользователей Интернет-ресурсов, как гендерная принадлежность, возраст, политическая ориентация, этническая принадлежность и географическое положение.

Задача определения социо-демографических атрибутов успешно решается методами машинного обучения. При этом наиболее часто используются признаки, извлекаемые из текстов сообщений пользователей Интернет-ресурсов, т. к. этот тип признаков является универсальным практически для любого ресурса. Добавление специфических для ресурса признаков позволяет улучшить качество определения атрибутов.

Рассмотренные текстовые признаки можно разделить на две основные группы. Признаки первой группы являются независимыми от языка сообщений. В эту группу входят структурные признаки и N-граммы символов

и слов. Признаки второй группы в общем случае зависят от языка. Это различные социолингвистические признаки. Любая система, использующая для определения социо-демографических атрибутов зависящие от языка признаки, также является зависящей от языка.

Недостатки существующих подходов:

1. ограниченный набор поддерживаемых атрибутов пользователя (большинство методов ограничивается определением пола и возраста);
2. отсутствие или недостаточная функциональность автоматических средств для сбора корпусов сообщений пользователей интернета и разметки их с помощью социо-демографических атрибутов пользователей (сообщения собираются и размечаются, в основном, вручную, что накладывает ограничения на размер корпуса и достоверность разметки);
3. отсутствие или недостаточная функциональность методов фильтрации зашумленных недостоверных данных (спама и ложных аккаунтов пользователей);
4. недостаточное использование социолингвистических методов для определения признаков, специфических для отдельных атрибутов и их значений;
5. недостаточное обоснование выбора используемых методов машинного обучения (извлечение признаков, отбор высокоинформативных признаков, обучение, классификация), что накладывает ограничение на качество результатов;
6. отсутствие методов, позволяющих осуществлять моделирование пользовательских атрибутов не по отдельности, а в зависимости от других атрибутов, что накладывает ограничение на качество результатов;
7. немногочисленные открытые программные реализации и веб-сервисы ограничены по функционалу (определяются только пол и/или возраст пользователя), а также в силу недостаточного качества реализации непригодны для интегрирования с промышленными приложениями;
8. отсутствие открытых программных реализаций и/или веб-сервисов для русского языка.

3. Метод

Абсолютное большинство современных методов определения демографических атрибутов пользователей основаны на применении методов машинного обучения с учителем с целью классификации пользователей по лингвистическим и другим признакам в предопределённые классы, соответствующие различным значениям изучаемых атрибутов. Сообщения пользователя рассматриваются как набор символьных строк, из которых извлекаются признаки, а для разметки применяются дополнительные источники данных о пользователе, причём в большинстве случаев разметка производится вручную.

Разработанный метод обладает следующими преимуществами:

- автоматическое построение исходного набора данных;

- извлечение большого количества признаков различных типов из текстов сообщений пользователей Twitter;
- расширяемый набор поддерживаемых атрибутов: все поля Facebook-профиля, а также любая информация о предпочтениях и интересах пользователя могут быть использованы в качестве атрибутов;
- расширяемый набор поддерживаемых языков благодаря использованию автоматической идентификации языка текста сообщений и применению метода построения исходного набора данных, не зависимого от языка.

Метод состоит из следующих этапов:

- построение исходного набора данных;
- предварительная обработка текста;
- построение признакового описания;
- обучение;
- классификация.

Все этапы, за исключением первого, выполняются отдельно для каждого атрибута.

На этапе **построения исходного набора данных** производится сбор данных пользователей из сети Twitter. Для каждого пользователя сначала запрашивается только его профиль в сети Twitter. При наличии в нём ссылки на профиль того же пользователя в сети Facebook (в которой набор пользовательских атрибутов существенно больше, чем в Twitter) запрашиваются и сохраняются все доступные сообщения пользователя из сети Twitter. После чего для текущего пользователя запрашивается и сохраняется его профиль в сети Facebook, из которого извлекаются указанные пользователем значения его атрибутов.

Таким образом, элементом набора данных для каждого атрибута и языка является набор символьных строк, полученных из текстов сообщений и профиля одного пользователя в Twitter, а также значение атрибута у данного пользователя в Facebook.

На этапе **предварительной обработки текста** к текстам полученного на предыдущем этапе набора данных применяется метод определения языковой принадлежности текста (библиотека *language-detection*). После этого данные пользователей распределяются в различные наборы данных в зависимости от языка пользователя.

Предварительно осуществляется фильтрация сообщений, авторство которых не принадлежит пользователю (*ретвиты*). Поскольку цитирование сообщений других пользователей является весьма популярным способом распространения информации в сети Twitter, этот шаг предварительной обработки особенно важен для повышения точности метода.

На этапе **построения признакового описания** из сообщений пользователей извлекаются лингвистические признаки. Из полученных токенов строится набор признаков в виде N-грамм размером от 1 до 3 с учётом порядка токенов. Каждый тип признаков представлен двумя подтипами: с учётом и без учёта регистра символов.

Итоговый вектор признаков для пользователя является бинарным, то есть содержит только информацию о наличии или отсутствии признака в его текстовых данных. Количество экземпляров одного признака игнорируется.

На этапе **обучения** производится построение модели классификации с использованием алгоритма SVM (машины опорных векторов).

На этапе **классификации** в качестве входных данных используются тексты сообщений и поля профиля произвольного пользователя. Выполняется алгоритм классификация для заданного языка и атрибута. Результатом является значение атрибута выбранного пользователя.

4. Результаты экспериментов

Общая схема экспериментальных исследований для каждого демографического атрибута следующая:

1. Получить размеченные данные о пользователях, содержащие корректное значение исследуемого атрибута.
2. Обучить модель классификатора на части размеченных данных, а именно на случайном подмножестве пользователей мощностью 90% от исходного множества.
3. Классифицировать с помощью обученной модели остальную часть размеченных данных.
4. Оценить качество классификации путем сравнения значений исследуемого атрибута в размеченных данных и в результате классификации.

Размеченные данные о пользователях, содержащие корректное значение исследуемого социо-демографического атрибута, извлекаются на этапе сбора и разметки сообщений пользователей. Объем извлекаемых данных: 500 пользователей для каждого исследуемого социо-демографического атрибута и поддерживаемого языка. Таким образом, обучение происходит на 450 случайных пользователях, тестирование – на оставшихся 50 пользователях.

В качестве оценки качества используется общепринятая метрика точности (*Accuracy*).

В таблице 1 приведены результаты экспериментов при использовании N-грамм порядка три.

Таблица 1 Результаты экспериментальных исследований

Тестовая задача	**Язык**	**Точность, %**
Определение пола	английский	84
	русский	86
	испанский	94
	немецкий	88
	французский	94
	итальянский	82
Определение возраста	английский	94
	русский	92
	испанский	92
	немецкий	80
	французский	84
	итальянский	94
Определение семейного положения	английский	98
	русский	96
	испанский	98
	немецкий	94
	французский	98
	итальянский	94
Определение уровня образования	английский	92
Определение религиозных взглядов	английский	94
Определение политических взглядов	английский	82

5. Заключение

Непосредственной областью применения предложенного метода является интернет-маркетинг: повышение точности таргетированного продвижения товаров и услуг в интернете позволит повысить результативность рекламных кампаний и в конечном итоге увеличить прибыль производителей и посредников.

Использование метода в политических целях позволит собирать дополнительную информацию об избирателях и более эффективно расходовать средства рекламных кампаний.

Применение метода органами правопорядка позволит производить частичную де-анонимизацию преступников путём анализа их сообщений.

Ряд задач прикладного характера может быть решён напрямую с использованием разработанного метода, в частности:

- система рекомендаций товаров и услуг для интернет-магазина с учётом демографических профилей их пользователей, построенных путём анализа текстов их сообщений/отзывов;
- система рекомендаций пользователей для установления связей дружбы/следования в социальных сетях с учётом похожести демографических профилей различных пользователей, построенных путём анализа текстов их сообщений;
- система рекомендаций телепередач с учётом демографический профилей пользователей средств массовой информации, построенных путём анализа их отзывов и сообщений о различных СМИ.

Также разработанный задел открывает возможность для проведения исследований по различным направлениям в области обработки персональных данных пользователей сети Интернет, например, в ходе анализа социальных сетей (поиск сообществ и кластеризация социального графа), персонализации информационного поиска (вывод более точного поискового результата), упрощение решения задач компьютерной лингвистики (устранение многозначности и поиск омонимов во время исследования сообщений пользователей).

Список литературы

[1]. Sloan L. Knowing the Tweeters: Deriving Sociologically Relevant Demographics from Twitter. [Текст] / L. Sloan [et al.] – Sociological Research Online. – 2013. – Т. 18. – №. 3. – p. 7.

[2]. Tang C. What’s in a name: A study of names, gender inference, and gender behavior in facebook. [Текст] / C. Tang [et al.] – Database Systems for Adanced Applications. – Springer Berlin Heidelberg, 2011. – pp. 344–356.

[3]. Miller Z. Gender Prediction on Twitter Using Stream Algorithms with N-Gram Character Features. [Текст] / Z. Miller, B. Dickinson, W. Hu – International Journal. – 2012. – Т. 2.
[4]. Deitrick W. Gender identification on twitter using the modified balanced winnow. [Текст] / W. Deitrick [et al.] – Communications and Network. – 2012. – Т. 4. – №. 3. – pp. 189–195.
[5]. Burger J. D. Discriminating gender on Twitter. [Текст] / J. D. Burger [et al.] – Proceedings of the Conference on Empirical Methods in Natural Language Processing. – Association for Computational Linguistics, 2011. – pp. 1301–1309.
[6]. Schwartz H. A. Personality, Gender, and Age in the Language of Social Media: The Open-Vocabulary Approach. [Текст] / H. A. Schwartz [et al.] – PloS one. – 2013. – Т. 8. – №. 9. – p. 73791.
[7]. Filippova K. User demographics and language in an implicit social network. [Текст] – Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. – Association for Computational Linguistics, 2012. – pp. 1478–1488.
[8]. Cheng N. Author gender identification from text. [Текст] / N. Cheng, R. Chandramouli, K. P. Subbalakshmi – Digital Investigation. – 2011. – Т. 8. – №. 1. – pp. 78–88.
[9]. Rao D. Classifying latent user attributes in twitter. [Текст] / D. Rao [et al.] – Proceedings of the 2nd international workshop on Search and mining user-generated contents. – ACM, 2010. – pp. 37–44.
[10]. Rao D. Hierarchical Bayesian Models for Latent Attribute Detection in Social Media. [Текст] / D. Rao [et al.] – ICWSM. – 2011.
[11]. Mukherjee A. Improving gender classification of blog authors. [Текст] / A. Mukherjee, B. Liu – Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing. – Association for Computational Linguistics, 2010. – pp. 207–217.
[12]. Liu W. What's in a Name? Using First Names as Features for Gender Inference in Twitter. [Текст] / W. Liu, D. Ruths – 2013 AAAI Spring Symposium Series. – 2013.
[13]. Al Zamal F. Homophily and Latent Attribute Inference: Inferring Latent Attributes of Twitter Users from Neighbors. [Текст] / F. Al Zamal, W. Liu, D. Ruths – ICWSM. – 2012.
[14]. Garera N. Modeling latent biographic attributes in conversational genres. [Текст] / N. Garera, D. Yarowsky – Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP. – Association for Computational Linguistics, 2009. – Vol. 2, pp. 710–718.
[15]. Schler J. Effects of Age and Gender on Blogging. [Текст] / J. Schler [et al.] – AAAI Spring Symposium: Computational Approaches to Analyzing Weblogs. – 2006. – pp. 199–205.
[16]. Goswami S. Stylometric analysis of bloggers' age and gender. [Текст] / S. Goswami, S. Sarkar, M. Rustagi – Third International AAAI Conference on Weblogs and Social Media. – 2009.
[17]. Nguyen D. Author age prediction from text using linear regression. [Текст] / D. Nguyen, N. A. Smith, C. P. Rosé – Proceedings of the 5th ACL-HLT Workshop on Language Technology for Cultural Heritage, Social Sciences, and Humanities. – Association for Computational Linguistics, 2011. – pp. 115–123.
[18]. van Heerden C. Combining regression and classification methods for improving automatic speaker age recognition. [Текст] / C. van Heerden [et al.] – Acoustics Speech

and Signal Processing (ICASSP), 2010 IEEE International Conference on. – IEEE, 2010. – pp. 5174–5177.
[19]. Nguyen D. “How Old Do You Think I Am?”: A Study of Language and Age in Twitter. [Текст] / D. Nguyen [et al.] – Seventh International AAAI Conference on Weblogs and Social Media. – 2013.
[20]. Peersman C. Predicting age and gender in online social networks. [Текст] / C. Peersman, W. Daelemans, L. Van Vaerenbergh – Proceedings of the 3rd international workshop on Search and mining user-generated contents. – ACM, 2011. – pp. 37–44.
[21]. Rosenthal S. Age prediction in blogs: A study of style, content, and online behavior in pre-and post-social media generations. [Текст] / S. Rosenthal, K. McKeown. – Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies. – Association for Computational Linguistics, 2011. – Vol. 1, pp. 763–772.
[22]. Pennacchiotti M. A Machine Learning Approach to Twitter User Classification. [Текст] / M. Pennacchiotti, A. M. Popescu – ICWSM. – 2011.
[23]. Conover M. D. Predicting the political alignment of twitter users. [Текст] / M. D. Conover [et al.] – Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom). – IEEE, 2011. – pp. 192–199.
[24]. Eisenstein J. A latent variable model for geographic lexical variation [Текст] / J. Eisenstein [et al.] – Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing. – Association for Computational Linguistics, 2010. – pp. 1277–1287.
[25]. **Cheng Z.** You are where you tweet: a content-based approach to geo-locating twitter users. [Текст] / Z. Cheng, J. Caverlee, K. Lee – Proceedings of the 19th ACM international conference on Information and knowledge management. – ACM, 2010. – pp. 759–768.
[26]. **Al Zamal F., Liu W., Ruths D.** Homophily and Latent Attribute Inference: Inferring Latent Attributes of Twitter Users from Neighbors //ICWSM. – 2012.
[27]. **Rao D. et al**. Classifying latent user attributes in twitter //Proceedings of the 2nd international workshop on Search and mining user-generated contents. – ACM, 2010. – С. 37-44.
[28]. **Burger J. D. et al.** Discriminating gender on Twitter //Proceedings of the Conference on Empirical Methods in Natural Language Processing. – Association for Computational Linguistics, 2011. – С. 1301-1309.
[29]. **Kótyuk G., Buttyán L**. A machine learning based approach for predicting undisclosed attributes in social networks //Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on. – IEEE, 2012. – С. 361-366.
[30]. **Caruana, G**. A survey of emerging approaches to spam filtering [Текст] / G. Caruana, M. Li // ACM Computing Surveys (CSUR), Vol. 44, No.2, February 2012. pp. 1-27.
[31]. **Stafford, G.** An Evaluation of the Effect of Spam on Twitter Trending Topics [Электронный ресурс] — Электрон. дан. - США, [2013] — Режим доступа: http://homepages.gac.edu/~lyu/Grant_paper.pdf, свободный. — Англ.
[32]. **Martinez-Romo, J.** Detecting malicious tweets in trending topics using a statistical analysis of language [Текст] / J. Martinez-Romo, L. Araujo // Expert Systems with Applications, Vol. 40, No.8, June 2013. pp. 2992-3000.

[33]. **Almeida, T. A.** Advances in spam filtering techniques. In Computational Intelligence for Privacy and Security [Текст] / T. A. Almeida, A.Yamakami // Computational Intelligence for Privacy and Security, Vol. 394, 2012. pp. 199-214.
[34]. **Wang, A. H.** Machine Learning for the Detection of Spam in Twitter Networks [Текст] / A. H. Wang // e-Business and Telecommunications, Vol. 222, 2012. pp. 319-333.
[35]. **Ahmed, F.** Generic Statistical Approach for Spam Detection [Текст] / F.Ahmed, M. A. Abulaish // Computer Communications, Vol. 36, June 2013. pp. 1120-1129.
[36]. **Thomas, K.** Suspended Accounts in Retrospect: An Analysis of Twitter Spam [Текст] / K. Thomas, C. Grier, V. Paxson, D. Song // Proceedings of the Internet Measurement Conference 2011 (IMC 2011) , Berlin, Germany, November 2-4. 2011. pp. 243-258.
[37]. **Sridharan, V.** Twitter games: how successful spammers pick targets [Текст] / V. Sridharan, V. Shankar, M. Gupta // Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, Florida, USA, December 3-7. 2012. pp. 389-398.
[38]. **Левенштейн, В.И.** Двоичные коды с исправлением выпадений, вставок и замещений символов [Текст] / В.И. Левенштейн // Доклады Академий Наук СССР, 1965, Т. 163, №4. С. 845-848.
[39]. **Lin P.C.** A study of effective features for detecting long-surviving Twitter spam accounts [Текст] / P.C. Lin, P.M. Huang // The 15th International Conference on Advanced Communications Technology, Phoenix Park, PyeongChang, South Korea, January 27-30. 2013. pp. 841 - 846
[40]. **Романов А.С., Мещеряков Р.В.** *Определение пола автора короткого электронного сообщения* // Компьютерная лингвистика и интеллектуальные технологии: По материалам ежегодной Междунар. конф. «Диалог» (Бекасово, 25–29 мая 2011 г.). М. : Изд-во РГГУ, 2011. Вып. 10 (17). С. 620–626

Detection of demographic attributes of microblog users

Anton Korshunov, Ivan Beloborodov, Andrey Gomzin, Christina Chuprina, Nikita Astrakhantsev, Yaroslav Nedumod, Denis Turdakov
{korshunov, ivbel, gomzin, chuprina, astrakhantsev, yaroslav.nedumov, turdakov}@ispras.ru
ISP RAS, Moscow, Russia

Abstract. Users of internet services often make errors or intentionally provide misleading information about their demographic attributes, including gender, age, marital status, education, religious and political views. At the same time, knowing values of user attributes allows to enhance the performance of recommender systems, internet marketing solutions, and other applications based on personalized results. In the paper, a method is proposed for automatic detection of demographic attributes of Twitter users by analyzing their textual messages and other data from their profiles. The method is based on a machine learning algorithm trained with binary vectors of token N-grams extracted from user posts. Its distinctive features are fully automatic compilation of training and testing data sets as well as support for a broad and extendable range of languages and demographic attributes. This is achieved by exploiting Facebook accounts associated with user profiles in Twitter. Additional steps are detecting language of posts and filtering borrowed content. Experimental study showed high accuracy of gender, age, and marital status detection for the most popular languages: English, Russian, German, French, Italian, and Spanish. Besides, detection of education, religious and political views is also supported for English.

Keywords: demographic characteristics; demographic attributes; social networks; microblogs; natural language processing; content analysis; computational linguistics; machine learning.

References

[1]. Sloan L. et al. Knowing the Tweeters: Deriving Sociologically Relevant Demographics from Twitter. Sociological Research Online. – 2013. – Т. 18. – №. 3. – p. 7.
[2]. Tang C. et al. What's in a name: A study of names, gender inference, and gender behavior in facebook. Database Systems for Adanced Applications. – Springer Berlin Heidelberg, 2011. – pp. 344–356.
[3]. Miller Z., Dickinson B., Hu W. Gender Prediction on Twitter Using Stream Algorithms with N-Gram Character Features. International Journal. – 2012. – Т. 2.
[4]. Deitrick W. Gender identification on twitter using the modified balanced winnow. Communications and Network. – 2012. – Т. 4. – №. 3. – pp. 189–195.
[5]. Burger J. D. et al. Discriminating gender on Twitter. Proceedings of the Conference on Empirical Methods in Natural Language Processing. – Association for Computational Linguistics, 2011. – pp. 1301–1309.
[6]. Schwartz H. A. et al. Personality, Gender, and Age in the Language of Social Media: The Open-Vocabulary Approach. PloS one. – 2013. – Т. 8. – №. 9. – p. 73791.
[7]. Filippova K. User demographics and language in an implicit social network. Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language

Processing and Computational Natural Language Learning. – Association for Computational Linguistics, 2012. – pp. 1478–1488.
[8]. Cheng N., Chandramouli R., Subbalakshmi K. P. Author gender identification from text. Digital Investigation. – 2011. – T. 8. – №. 1. – pp. 78–88.
[9]. Rao D. et al. Classifying latent user attributes in twitter. Proceedings of the 2nd international workshop on Search and mining user-generated contents. – ACM, 2010. – pp. 37–44.
[10]. Rao D. et al. Hierarchical Bayesian Models for Latent Attribute Detection in Social Media. ICWSM. – 2011.
[11]. Mukherjee A., Liu B. Improving gender classification of blog authors. Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing. – Association for Computational Linguistics, 2010. – pp. 207–217.
[12]. Liu W. Ruths D. What's in a Name? Using First Names as Features for Gender Inference in Twitter. 2013 AAAI Spring Symposium Series. – 2013.
[13]. Al Zamal F., Liu W., Ruths D. Homophily and Latent Attribute Inference: Inferring Latent Attributes of Twitter Users from Neighbors. ICWSM. – 2012.
[14]. Garera N., Yarowsky D. Modeling latent biographic attributes in conversational genres. Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP. – Association for Computational Linguistics, 2009. – Vol. 2, pp. 710–718.
[15]. Schler J. et al. Effects of Age and Gender on Blogging. AAAI Spring Symposium: Computational Approaches to Analyzing Weblogs. – 2006. – pp. 199–205.
[16]. Goswami S., Sarkar S., Rustagi M. Stylometric analysis of bloggers' age and gender. Third International AAAI Conference on Weblogs and Social Media. – 2009.
[17]. Nguyen D. Smith N.A., Rosé C.P. Author age prediction from text using linear regression. Proceedings of the 5th ACL-HLT Workshop on Language Technology for Cultural Heritage, Social Sciences, and Humanities. – Association for Computational Linguistics, 2011. – pp. 115–123.
[18]. van Heerden C. et al. Combining regression and classification methods for improving automatic speaker age recognition. Acoustics Speech and Signal Processing (ICASSP), 2010 IEEE International Conference on. – IEEE, 2010. – pp. 5174–5177.
[19]. Nguyen D. et al. "How Old Do You Think I Am?": A Study of Language and Age in Twitter. Seventh International AAAI Conference on Weblogs and Social Media. – 2013.
[20]. Peersman C., Daelemans W., Van Vaerenbergh L. Predicting age and gender in online social networks. Proceedings of the 3rd international workshop on Search and mining user-generated contents. – ACM, 2011. – pp. 37–44.
[21]. Rosenthal S. McKeown K. Age prediction in blogs: A study of style, content, and online behavior in pre-and post-social media generations. Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies. – Association for Computational Linguistics, 2011. – Vol. 1, pp. 763–772.
[22]. Pennacchiotti M., Popescu A.M. A Machine Learning Approach to Twitter User Classification. ICWSM. – 2011.
[23]. Conover M. D. Predicting the political alignment of twitter users. Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom). – IEEE, 2011. – pp. 192–199.

[24]. Eisenstein J. et al. A latent variable model for geographic lexical variation Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing. – Association for Computational Linguistics, 2010. – pp. 1277–1287.
[25]. Cheng Z., Caverlee J., Lee K. You are where you tweet: a content-based approach to geo-locating twitter users. Proceedings of the 19th ACM international conference on Information and knowledge management. – ACM, 2010. – pp. 759–768.
[26]. Al Zamal F., Liu W., Ruths D. Homophily and Latent Attribute Inference: Inferring Latent Attributes of Twitter Users from Neighbors. ICWSM. – 2012.
[27]. Rao D. et al. Classifying latent user attributes in twitter. Proceedings of the 2nd international workshop on Search and mining user-generated contents. – ACM, 2010. – S. 37-44.
[28]. Burger J. D. et al. Discriminating gender on Twitter. Proceedings of the Conference on Empirical Methods in Natural Language Processing. – Association for Computational Linguistics, 2011. – S. 1301-1309.
[29]. Kótyuk G., Buttyán L. A machine learning based approach for predicting undisclosed attributes in social networks. Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on. – IEEE, 2012. – S. 361-366.
[30]. Caruana, G., Li M. A survey of emerging approaches to spam filtering. ACM Computing Surveys (CSUR), Vol. 44, No.2, February 2012. pp. 1-27.
[31]. Stafford G., Yu L. L. An Evaluation of the Effect of Spam on Twitter Trending Topics. Social Computing (SocialCom), 2013 International Conference on. – IEEE, 2013. – C. 373-378.
[32]. Martinez-Romo, J., Araujo L. Detecting malicious tweets in trending topics using a statistical analysis of language. Expert Systems with Applications, Vol. 40, No.8, June 2013. pp. 2992-3000.
[33]. Almeida T. A., Yamakami A. Advances in spam filtering techniques. Computational Intelligence for Privacy and Security. – Springer Berlin Heidelberg, 2012. – C. 199-214.
[34]. 34. Wang, A. H. Machine Learning for the Detection of Spam in Twitter Networks. e-Business and Telecommunications, Vol. 222, 2012. pp. 319-333.
[35]. Ahmed, F., Abulaish M.A. Generic Statistical Approach for Spam Detection. Computer Communications, Vol. 36, June 2013. pp. 1120-1129.
[36]. Thomas, K., Grier C., Paxson V., Song D. Suspended Accounts in Retrospect: An Analysis of Twitter Spam. Proceedings of the Internet Measurement Conference 2011 (IMC 2011) , Berlin, Germany, November 2-4. 2011. pp. 243-258.
[37]. Sridharan V., Shankar V., Gupta M. Twitter games: how successful spammers pick targets. Proceedings of the 28th Annual Computer Security Applications Conference, Orlando, Florida, USA, December 3-7. 2012. pp. 389-398.
[38]. Levenshtejn V.I. Dvoichnye kody s ispravleniem vypadenij, vstavok i zameshhenij simvolov [Binary codes with correction for deletions, insertions, and substitutions of characters]. Doklady Akademij Nauk SSSR [The Proceedings of the USSR Academy of Sciences], 1965, T. 163, №4. C. 845-848. (in Russian)
[39]. Lin P.C., Huang P.M. A study of effective features for detecting long-surviving Twitter spam accounts. The 15th International Conference on Advanced Communications Technology, Phoenix Park, PyeongChang, South Korea, January 27-30. 2013. pp. 841 — 846
[40]. Romanov A.S., Meshheryakov R.V. Opredelenie pola avtora korotkogo ehlektronnogo soobshheniya [Gender identification of the author of a short message]. Komp'yuternaya

lingvistika i intellektual'nye tekhnologii: Po materialam ezhegodnoj Mezhdunar. konf. «Dialog» (Bekasovo, 25–29 maya 2011 g.). [Computational Linguistics and Intellectual Technologies: papers from the Annual conference “Dialogue” (Bekasovo, May 25-29, 2011)] Moscow.: RGGU, 2011.Issue 10 (17) – Moscow, RGGU, 2011. pp. 620–626. (in Russian)

Нахождение корней систем алгебраических уравнений с помощью базиса Гребнера

А.В. Шокуров, shok@ispras.ru
ИСП РАН

Аннотация. Описан и обоснован алгоритм нахождения решения системы алгебраических уравнений над полем k для идеалов нулевой размерности, в случае если задан базис Гребнера идеала этой системы для лексикографического порядка на термах от ее переменных. Полученное решение лежит в алгебраическом замыкании основного поля. Приведен пример системы алгебраических уравнений, имеющей единственное решение в основном поле, а общее число решений экспоненциально относительно описания этой системы.

Ключевые слова. Базис Гребнера, идеал,

1. Введение

Пусть k — поле, а K — его алгебраическое замыкание. Напомним, что алгебраическое замыкание кольца рациональных функций над полем κ от бесконечного числа независимых переменных называется универсальным расширением поля K. Будем обозначать его Ω. Идеал, порожденный конечным множеством $F \subseteq k[x_1, \dots, x_n]$, обозначим через (F). Согласно теореме Гильберта о базисе, для любого идеала существует конечное множество многочленов, порождающее этот идеал.

Многообразием идеала (см. [1]) $I \subseteq k[x_1, \dots, x_n]$ в K^n будем называть множество

$$V(I) = \{\xi \in K^n \mid \forall p \in I \text{ выполняется равенство } p(\xi) = 0\}.$$

Элемент $\xi \in \Omega^n$ называется общим корнем простого идеала I, если выполнены условия:

- $\xi \in V(I)$
- $p \in I \Leftrightarrow p(\xi) = 0.$

Задача 1. Задано конечное множество *F* элементов в кольце многочленов над полем и базис Гребнера *G* идеала (F). Определить, что выполняется:

- $V\big((F)\big) = \emptyset$, или
- $V((F)) \neq \emptyset$ и конечно, или
- $V((F)) \neq \emptyset$ и бесконечно.

Задача 2. Задано конечное множество *F* элементов в кольце многочленов над полем и базис Гребнера *G* идеала *(F).* Определить, что выполняется:

- $dim_k\,(F) = 0$ или
- $dim_k\,(F) \neq 0.$

Напомним определение лексикографического порядка на множестве термов от переменных $x_1, \dots, x_n$. Поскольку имеется взаимнооднозначное соответствие множества термов от переменных $x_1, \dots, x_n$ и элементами прямого произведения *n* экземпляров множества неотрицательных чисел $\mathbb{Z}^n$, достаточно определить порядок на $\mathbb{Z}_+^n$. Для $\alpha, \beta \in \mathbb{Z}_+^n$ будем считать, что $\alpha \succ \beta$, если $\alpha \neq \beta$ и при некотором $1 < i_0 \leq n$

- $\alpha_{i_0} > \beta_{i_0}$
- $\alpha_{i_0} = \beta_{i_0}$ при любом $n \geq i \geq i_0.$

В частности, $x_1 \prec x_2 \prec \cdots \prec x_n$.

Задача 3. Задано конечное множество F элементов в кольце многочленов $k[x_1, \dots, x_n]$, для которого $dim_k\,(F) = 0$, и базис Гребнера G идеала (F) относительно лексикографического порядка. Найти (построить) множество V((F)).

Приводятся алгоритмы решения поставленных задач и доказана их корректность.

2. Идеалы нулевой размерности

Лемма 1. Многообразие V(I) решений идеала $I \subseteq k[x_1, \dots, x_n]$ конечно тогда и только тогда, когда $k[x_1, \dots, x_n]/I$ — конечномерное над k векторное пространство.

Доказательство. **Необходимость.** Если решений нет, то согласно теореме Гильберта о нулях идеал I содержит единицу и, поэтому, совпадает с кольцом многочленов $k[x_1, \dots, x_n]$. Следовательно,

$$dim_k\ k[x_1, \dots, x_n]/I = 0.$$

Пусть теперь множество *V(I)* непусто, конечно и $(\lambda_{i,1}, \dots, \lambda_{i,n})$, при $i = 1, \dots, m$, — все его элементы. Поскольку $\lambda_{i,j}$ принадлежат алгебраическому замыканию поля k, то для каждого такого $\lambda_{i,j}$ существует многочлен $p_{\lambda_{i,j}}(x) \in k[x]$ с корнем $\lambda_{i,j}$. Тогда многочлены

$$p_j(x_j) = \prod_{i=1}^{m} p_{i,j}(x_j) \in k[x_1, \dots, x_n], \quad j = 1, \dots, n$$

обращаются в ноль на всех решениях идеала, и, следовательно, по теореме Гильберта о нулях существуют такие k_j, что $p_j^{k_j} \in I$. Поэтому,

$$deg_k\, k[x_1, \dots, x_n]/I \le \prod_{i=1}^{m}(m_j\, k_j + 1),$$

где $m_j = \deg p_j$.

Достаточность. Пусть теперь $dim_k\ k[x_1, \dots, x_n]/I$ конечна. Если эта размерность нулевая, то $I = k[x_1, \dots, x_n]$ и, следовательно, множество решений пусто, т.е. конечно.

Рассмотрим случай когда размерность $dim_k\ k[x_1, \dots, x_n]/I$ конечна, но не равна нулю. Рассмотрим базис Гребнера идеала *I* для лексикографического порядка на множестве многочленов. Согласно определению базиса Гребнера, каждый многочлен из $k[x_1, \dots, x_n]$ редуцируется к единственному нередуцируемому многочлену, т.е. все термы полученного многочлена не содержат термы, делящиеся на старшие термы многочленов из базиса Гребнера идеала *I*. Рассмотрим многочлены вида $f(x_1)$. По определению лексикографического порядка, любой терм, в который входит хотя бы одна из переменных $x_2, \dots, x_n$, старше любого терма вида x_1^m. Поэтому, существует многочлен $p_1(x_1)$, принадлежащий базису Гребнера идеала *I* (в противном случае размерность векторного пространства $k[x_1, \dots, x_n]/I$ над полем *к* была бы бесконечной), а следовательно, и идеалу *I*. Аналогично для всех остальных переменных существуют $p_i(x_i) \in I$. Тогда все решения идеала *I* лежат в произведении всех решений $p_i(x_i) = 0$, т.е. в конечном множестве. □

Лемма 2. *Размерность идеала $I \subseteq k[x_1, \dots, x_n]$ равна нулю тогда и только тогда, когда многообразие V(I) конечно и непусто.*

Доказательство. Пусть $I = [I_1, \dots, I_s]$ — неприводимое представление идеала *I* примарными идеалами и $p_1, \dots, p_s$ — ассоциированные с этим представлением простые идеалы (согласно теореме Ласкера, см. [1]). Размерность идеала *I*, по определению, равна максимальной из размерностей простых идеалов $p_1, \dots, p_s$.

Предположим, что $dim_k I = 0$. Тогда для всех $i = 1, \dots, s$ выполнено $dim_k p_i = 0$. Непосредственно из определений следует, что $V(p_i) = V(I_i)$ и, следовательно, множества $V(I_i)$ конечны. Поэтому и множество решений $V(I) = V(I_1) \cup \dots \cup V(I_s)$ конечно.

Пусть теперь множество *V(I)* — конечно. Тогда и все $V(p_i)$ конечны. Достаточно проверить, что для простого идеала *p* размерности большей нуля множество *V(p)* бесконечно. Для этого, согласно лемме 1, достаточно убедиться, что величина $dim_k\ k[x_1, \dots, x_n]/p$ бесконечна. Пусть $(\xi_1, \dots, \xi_n)$ — общий корень идеала *p*. Тогда определены вложения

$$k \subset k[\xi_1, \dots, \xi_n] \subset k(\xi_1, \dots, \xi_n) \subset \Omega.$$

Поскольку *dim* p > 0, компоненты общего корня этого идеала содержат трансцендентные элементы. Без ограничения общности можно считать, что ξ_1 трансцендентен. Тогда элементы $\xi_1, \xi_1^2, \dots, \xi_1^m, \dots$ — линейно независимы над *k*. Следовательно, $dim_k\ k[\xi_1, \dots, \xi_n]$ бесконечна, а поскольку, в силу определения общего корня $(\xi_1, \dots, \xi_n)$ простого идеала *p*, выполняется равенство $k[\xi_1, \dots, \xi_n] = k[x_1, \dots, x_n]/p$, то и величина $dim_k\ k[x_1, \dots, x_n]/p$ бесконечна.

Лемма 3. Пусть G — базис Гребнера идеала I. Отображение

$$\pi: k[x_1, \dots, x_n]/I \longrightarrow k[x_1, \dots, x_n]$$

заданное формулой $f + I \mapsto h$, где $f \to_{G^*} \underline{h}$ — неприводимая редукция, определено корректно, взаимно однозначно и является k-гомоморфизмом векторных пространств.

Доказательство. Формула $f \to_{G^*} \underline{h}$ задает гомоморфизм *k*-векторных пространств

$$\varphi: k[x_1, \dots, x_n] \longrightarrow k[x_1, \dots, x_n]$$

ядром которого является идеал *I*. Следовательно, π определено корректно и является *k*-мономорфизмом. □

Следствие 1. Пусть G — базис Гребнера идеала I. Размерность идеала I равна нулю тогда и только тогда, когда множество неприводимых относительно базиса Гребнера G термов конечно.

Теорема 1. Пусть G — базис Гребнера собственного идеала кольца многочленов $k[x_1, \dots, x_n]$. Этот идеал имеет размерность 0 тогда и только тогда, когда для любого допустимого порядка при каждом $1 \le i \le n$ существует многочлен $g_i \in G$ со старшим термом $x_i^{\nu_i}$ где ν_i — некоторое неотрицательное целое число.

Доказательство. Если для некоторого i не существует элемента базиса Гребнера идеала I со старшим термом $x_i^{\nu_i}$, то термы x_i^m неприводимы.

Следовательно, множество неприводимых термов бесконечно, и, поэтому, согласно следствию 1 размерность идеала I не равна нулю.

Пусть базис Гребнера идеала I содержит многочлены g_i, старшими термами которых являются $x_i^{\nu_i}$. В этом случае необходимым условием неприводимости терма t является выполнение соотношений $deg_{x_i} t < \nu_i$ для всех i=1,...,n. Поэтому, мощность множества неприводимых термов не превосходит величину $\nu_1 \cdot \ldots \cdot \nu_n$, и, следовательно, конечна. А тогда, согласно следствию 1, размерность идеала I равна нулю. □

3. Решение систем уравнений

Решение задачи 1. Согласно теореме Гильберта о нулях, условие $V\big((F)\big) = \emptyset$ эквивалентно условию $1 \in (F)$, или, эквивалентно, $1 \in G$.

Пусть теперь $1 \notin G$. Тогда $V((F)) \neq \emptyset$. Вопрос о конечности или бесконечности многообразия *V((F))* решается теперь теоремой 1 и леммой 2. Достаточно преверить, содержит ли базис Гребнера *G* многочлены со старшими термами $x_i^{\nu_i}$ при всех *i* = 1, ..., *n*.

Задача 2 полностью решается в теореме 1.

Для решения **задачи 3** потребуется решить следующую задачу.

Задача нахождения хотя бы одного решения идеала нулевой размерности, если задан приведенный базис Гребнера этого идеала относительно лексикографического порядка. Предположим, что задан базис Гребнера $g_1, \ldots, g_m$ идеала $I \subseteq k[x_1, \ldots, x_n]$ нулевой размерности. Пусть также имеется оракул $\mathcal{A}$, решающий задачу нахождения корня любого многочлена одной переменной над *K*, где *K* — алгебраическое замыкание поля *k*.

Отметим, что имея базис Гребнера относительно некоторого порядка, всегда можно найти соответствующий базис Гребнера относительно лексикографического порядка (см., например, [3])

Решение задачи нахождения хотя бы одного решения идеала нулевой размерности, если задан его приведенный базис Гребнера относительно лексикографического порядка.

Пусть $I \subseteq k[x_1, \ldots, x_n]$ — идеал и G — приведенный базис Гребнера относительно лексикографического порядка этого идеала. Тогда пересечение $G \cap k[x_1]$ состоит в точности из одного многочлена $f(x_1) \in k[x_1]$ и является базисом Гребнера идеала $I_1 = I \cap k[x_1]$ кольца $k[x_1]$. Находим с помощью оракула $\mathcal{A}$ решение $\xi_1 \in K$ уравнения $f(x_1) = 0$ (*K* — алгебраическое замыкание поля *k*). Заметим, что для любого решения $(x_1^0, \ldots, x_n^0)$ идеала I выполняется соотношение $f(x_1^0) = 0$.

Предположим теперь, что найдено решение $(\xi_1, \ldots, \xi_i)$ идеала $I_i = I \cap k[x_1, \ldots, x_i]$. Чтобы найти продолжение $(\xi_1, \ldots, \xi_i, \xi_{i+1})$ полученного выше решения, вычислим элементы базиса Гребнера *G*, находящиеся в кольце

$k[x_1, \dots, x_{i+1}]$. Затем выполним подстановки $x_j = \xi_j$ для всех j=1,…, i в полученные многочлены базиса Гребнера. Получим набор многочленов, зависящих только от одной переменной x_{i+1}. Вычислим их наибольший общий делитель $g(x_{i+1})$. Как будет показано ниже, полученный многочлен имеет степень не менее единицы и, следовательно, имеет непустое множество решений в алгебраическом замыкании поля k. Находим с помощью оракула $\mathcal{A}$ решение $\xi_{i+1} \in K$ уравнения $g(x_{i+1}) = 0$. Тогда вектор $(\xi_1, \dots, \xi_i, \xi_{i+1})$ является решением идеала $I_{i+1} = I \cap k[x_1, \dots, x_{i+1}]$.

Далее повторяем описанную процедуру до тех пор, пока не найдем полный вектор решения идеала I.

Ниже приведен алгоритм для описанной процедуры нахождения решения алгебраической системы уравнений.

Алгоритм А. Дано: Базис Гребнера $G = (g_1, \dots, g_m)$ относительно лексикографического порядка идеала $I \subseteq k[x_1, \dots, x_n]$ нулевой размерности.

Выход: Точка $(x_1^0, \dots, x_n^0) \in K^n$, где K алгебраическое замыкание поля k.

Шаг 1 i := 1.

Шаг 2 Находим пересечение $G_1 = G \cap k[x_1]$, состоящее в точности из одного многочлена $g(x_1)$.

Шаг 3 x_i^0 := $\mathcal{A}(g(x_i))$ — некоторое решение уравнения $g(x_i) = 0$.

Шаг 4 i := i + 1.

Шаг 5 Если $i > n$, перейти к шагу 10.

Шаг 6 Находим пересечение $G_i = G \cap k[x_1, \dots, x_i]$.

Шаг 7 $G_i(x_1^0, \dots, x_{i-1}^0) := \{g(x_1^0, \dots, x_{i-1}^0, x_i) \mid g \in G_i\} \subseteq K[x_i]$.

Шаг 8 Находим $g(x_i)$ — наибольший общий делитель элементов множества $G_i(x_1^0, \dots, x_{i-1}^0)$.

Шаг 9 Переходим к шагу 3.

Шаг 10 Выход: Точка $(x_1^0, \dots, x_n^0) \in K^n$, где K алгебраическое замыкание поля k, является решением идеала I.

Теорема 2. Для любого идеала размерности ноль алгоритм A находит некоторое его решение.

Для доказательства теоремы достаточно доказать, что на шаге 8 приведенного выше алгоритма всегда получаем многочлен степени не меньше единицы, или, эквивалентно, каждое решение $(\xi_1, \dots, \xi_i) \in K^i$ идеала $I_i = I \cap k[x_1, \dots, x_i]$ продолжается до решения $(\xi_1, \dots, \xi_{i+1}) \in K^{i+1}$ идеала $I_{i+1} = I \cap k[x_1, \dots, x_{i+1}]$. Доказательство этого утверждения потребует несколько вспомогательных утверждений.

Для любого подмножества *M* кольца $k[x_1, \dots, x_n]$ и любого $0 < i < n$ будем использовать следующее **обозначение:** $M_i = M \cap k[x_1, \dots, x_i]$. Заметим, что если *I* — идеал кольца $k[x_1, \dots, x_n]$, *то* I_i — идеал кольца $k[x_1, \dots, x_i]$.

Лемма 4. Если G — приведенный базис Гребнера относительно лексикографического порядка идеала $I \subseteq k[x_1, \dots, x_i]$, то d — приведенный базис Гребнера идеала I_i в кольце многочленов от переменных $x_1, \dots, x_i$ относительно лексикографического порядка на термах.

Доказательство. Следует из определения базиса Гребнера для лексикографического порядка. □

Лемма 5. Пусть $I \subseteq k[x_1, \dots, x_n]$ — простой идеал. Тогда для любого $0 < i <$ n идеал I_i простой.

Доказательство. Действительно, если $pq \in I_i$, то тем более $pq \in I$, и, *следовательно*, $p \in I$ или $q \in I$. Поскольку $pq \in k[x_1, \dots, x_i]$, то и $p \in k[x_1, \dots, x_i]$ и $q \in k[x_1, \dots, x_i]$. Поэтому, $p \in I_i$ или $q \in I_i$. □

Аналогично доказывается

Лемма 6. Пусть $I \subseteq k[x_1, \dots, x_n]$ — примарный идеал. Тогда для любого $0 <$ i < n идеал I_i примарный.

Лемма 7. Пусть $I \subseteq k[x_1, \dots, x_n]$ — примарный идеал и J — ассоциированный с ним простой идеал. Тогда множества корней идеалов I и J совпадают.

Доказательство. Следует непосредственно из определения ассоциированного простого идеала примарного идеала. □

Лемма 8. Пусть $I \subseteq k[x_1, \dots, x_n]$ — примарный идеал и J — ассоциированный с ним простой идеал. Тогда простой идеал J_i ассоциирован с идеалом I_i.

Доказательство. Следует из лемм 5 и 6 и определения ассоциированного простого идеала примарного идеала. □

Лемма 9. *Если* $I \subseteq k[x_1, \dots, x_n]$ *— идеал размерности 0, то* I_m *является идеалом размерности 0 в кольце многочленов* $k[x_1, \dots, x_m]$ для всех m=1,…,n.

Доказательство. Пусть *G* — приведенный базис Гребнера идеала *I* размерности ноль. Согласно теореме 1, для всех $i = 1, \dots, n$ существуют многочлены $g_i \in G$ со старшими термами $x_i^{\nu_i}$. Если *G* — базис Гребнера

идеала I относительно лексикографического порядка, то $g_i \in G_i$ и для любого $1 \le i \le m$ элементы $g_i \in I_m$. Поскольку старший терм g_i равен $x_i^{v_i}$, то, по теореме 1, размерность идеала I_m равна нулю. □

Следствие 2. *Пусть G — базис Гребнера относительно лексикографического порядка идеала $I \subseteq k[x_1, \dots, x_n]$ размерности 0. Тогда G_1 состоит в точности из одного многочлена $f \in k[x_1]$ положительной степени.*

Лемма 10. *Пусть $I \subseteq k[x_1, \dots, x_n]$ — простой идеал размерности 0 и $(\xi_1, \dots, \xi_i) \in K^i$ — корень идеала I_i. Тогда при $1 < i < n$ — 1 существует $\xi_{i+1} \in K$ такой, что $(\xi_1, \dots, \xi_{i+1}) \in K^{i+1}$— корень идеала I_{i+1}.*

Доказательство. Поскольку идеал I размерности 0, он не совпадает со своим кольцом и, следовательно, по теореме Гильберта имеет некоторый корень $(\omega_1, \dots, \omega_n)$ в K^n. В частности, $(\omega_1, \dots, \omega_i) \in K^i$, также как и $(\xi_1, \dots, \xi_i) \in K^i$, является корнем идеала I_i. Поскольку согласно леммам 5 и 9 идеал I_i прост и имеет нулевую размерность, а все корни простого идеала сопряжены, то имеется изоморфизм подполей поля K

$$\varphi: k(\omega_1, \dots, \omega_i) \longrightarrow k(\xi_1, \dots, \xi_i)$$

заданный соответствиями $\omega_j \mapsto \xi_j$ для всех $j = 1, \dots, i$.

Пусть G — базис Гребнера идеала I и многочлен $h \in k(\omega_1, \dots, \omega_n)[x_{i+1}]$ является наибольшим общим делителем многочленов $f(x_{i+1}) = g(\omega_1, \dots, \omega_i, x_{i+1})$, где g_i пробегает G_{i+1}. Поскольку $(\omega_1, \dots, \omega_{i+1})$ является корнем идеала I_{i+1}, элемент ω_{i+1} удовлетворяет соотношению $h(\omega_{i+1}) = 0$. Верно и обратное, для любого корня ζ уравнения $h(x) = 0$, точка $(\omega_1, \dots, \omega_i, \zeta)$ также корень идеала I_{i+1}, а поскольку размерность простого идеала I_{i+1} равна нулю, то эта точка также является общим корнем этого идеала. Поскольку число корней идеала размерности нуль конечно, то и число решений уравнения $h(xi+i) = 0$ не пусто и конечно. Поэтому степень многочлена h положительна.

Обозначим через $\tilde{h}$ наибольший общий делитель многочленов $f(x_{i+1}) = g(\xi_1, \dots, \xi_i, x_{i+1})$, где g пробегает G_{i+1}. Тогда согласно определению изоморфизма φ выполняется соотношение $\varphi^*(h) = \tilde{h}$ и степени многочленов h и $\tilde{h}$ совпадают. Следовательно, степень многочлена $\tilde{h}$ положительна. Поэтому уравнение $\tilde{h}(x_{i+1}) = 0$ всегда разрешимо в K. Пусть его решение ξ_{i+1}. Тогда $(\xi_1, \dots, \xi_{i+1}) \in K^{i+1}$ — корень идеала I_{i+1}.

Лемма 11. *Пусть $(\xi_1, \dots, \xi_n) \in \Omega^n$ — общий корень идеала простого идеала I. Тогда $(\xi_1, \dots, \xi_i) \in \Omega^i$ — общий корень простого идеала I_i.*

Доказательство. Обозначим через J_i — простой идеал, состоящий из всех многочленов кольца $k[x_1, \dots, x_i]$, обращающихся в ноль в точке $(\xi_1, \dots, \xi_i)$. Тогда точка $(\xi_1, \dots, \xi_i)$ является общим корнем идеала J_i.

Очевидно, что $J_i \supseteq I_i$, а поскольку точка $(\xi_1, \dots, \xi_n) \in \Omega^n$ является общим

корнем идеала I, то и $J_i \subseteq I$. Следовательно, $J_i \subseteq I_i$. Поэтому $J_i = I_i$ и $(\xi_1, \dots, \xi_i) \in \Omega^i$ — общий корень идеала I_i. □

Лемма 12. *Пусть $I \subseteq k[x_1, \dots, x_n]$ —примарный идеал и $(\xi_1, \dots, \xi_i) \in \Omega^i$ — общий корень простого идеала ассоциированного с идеалом I_i. Тогда существует $\xi_{i+1} \in \Omega$ такой, что $(\xi_1, \dots, \xi_{i+1}) \in \Omega^{i+1}$ — общий корень идеала I_{i+1}.*

Доказательство. Следует из лемм 8 и 11. □

Лемма 13. *Пусть $I \subseteq k[x_1, \dots, x_n]$ — примарный идеал размерности 0 и $(\xi_1, \dots, \xi_i) \in K^i$ — корень идеала I_i. Тогда существует $\xi_{i+1} \in K$ такой, что $(\xi_1, \dots, \xi_{i+1}) \in K^{i+1}$ — корень идеала I_{i+1}.*

Доказательство. Является частным случаем леммы 12, поскольку каждый корень простого идеала размерности ноль является его общим корнем. □

Напомним, что *V(I)* — многообразие идеала $I \subseteq k[x_1, \dots, x_n]$.

Лемма 14. *Если идеал I является пересечением идеалов q_j где j пробегает от 1 до m, то $V(I) = \bigcup_{j=1}^{m} V(q_j)$.*

Доказательство. Пусть $\xi \in \bigcup_{j=1}^{m} V(q_j)$. Тогда при некотором $1 \le j \le m$ выполняется $\xi \in V(q_j)$. А поскольку $I \subseteq q_j$, то и $\xi \in V(I)$. Следовательно, $V(I) \supseteq \bigcup_{j=1}^{m} V(q_j)$

Пусть теперь $\xi \notin \bigcup_{j=1}^{m} V(q_j)$. Тогда для всех $j = 1, ..., m$ существуют $p_j \in q_j$ для которых $p_j(\xi) \neq 0$. Поскольку все q_j являются идеалами, то произведение $p = \prod_{s=1}^{m} p_s$ является их общим элементом и, следовательно, принадлежит идеалу *I*. Элемент ξ не принадлежит многообразию *V(I)*, проскольку выполняется $p(\xi) = \prod_{s=1}^{m} p_s(\xi) \neq 0$. Следовательно, $V(I) \subseteq \bigcup_{j=1}^{m} V(q_j)$. □

Лемма 15. *Пусть $I \subseteq k[x_1, \dots, x_n]$ — идеал нулевой размерности и $(\xi_1, \dots, \xi_i) \in K^i$ — корень идеала I_i. Тогда существует $\xi_{i+1} \in K$ такой, что $(\xi_1, \dots, \xi_{i+1}) \in K^{i+1}$ — корень идеала I_{i+1}.*

Доказательство. Согласно теореме Ласкера (см. [1]) существует представление идеала I в виде пересечения конечного множества примарных идеалов

$$I = \bigcap_{j=1}^{m} q_i.$$

Напомним, что размерностью идеала называется максимальная из размерностей ассоциированных с его примарными компонентами простых идеалов. Поскольку *I* — идеал размерности ноль, то и все идеалы q_j также нулевой размерности. Очевидно, выполняется равенство

$$I_i = \bigcap_{j=1}^{m} q_{j,i} \qquad (1)$$

где $q_{j,i} = q_j \cap k[x_1, \dots, x_i]$ — примарные идеалы размерности ноль. Поэтому, согласно лемме 14, для всех $i = 1, \dots, n$ имеет место разложение

$$V(I_i) = \bigcup_{j=1}^{m} V(q_{j,i}). \qquad (2)$$

Поскольку $(\xi_1, \dots, \xi_i) \in K^i$ — корень идеала I_i, то из представления (2) следует, что при некотором j элемент $(\xi_1, \dots, \xi_i) \in K^i$ является корнем идеала $q_{j,i}$. Поэтому, по лемме 13 существует $\xi_{i+1} \in K$ такой, что $(\xi_1, \dots, \xi_{i+1}) \in K^{i+1}$ — корень идеала $q_{j,i+1\ 1}$, а, следовательно, согласно формулам (1) и (2), является корнем идеала I_{i+1}. □

Определение 1. *Размерностью системы алгебраических уравнений называется размерность соответствующего идеала этой системы. Системы алгебраических уравнений размерности ноль будем называть полными.*

Лемма 16. *Пусть $p(x) \in \mathbb{Q}[x]$ — многочлен с рациональными коэффициентами от одной переменной. Тогда уравнение p(x) = 0 алгоритмически разрешимо в поле рациональных чисел.*

Поскольку задача нахождения базиса Гребнера идеала I относительно произвольного порядка является алгоритмически разрешимой, а по теореме 1 для идеала размерности ноль для каждой переменной x_i существуют многочлены $f_i(x_i)$, зависящие только от этой переменной и принадлежащие идеалу I, то задача построения таких многочленов алгоритмически разрешима. Пусть X_i — множество рациональных решений уравнения $f_i(x_i) = 0$. Тогда все рациональные решения системы идеала I принадлежат произведению $X_1 \times \dots \times X_n$. Поэтому из леммы 16 следует алгоритмическая разрешимость полной системы алгебраических уравнений над полем $\mathbb{Q}$.

Следствие 3. *Задача нахождения решения полной системы алгебраических уравнений над полем рациональных чисел является алгоритмически разрешимой.*

Для неполных систем уравнений, например, диафантовых уравнений утверждение следствия 3 не получается.

4. Пример системы алгебраических уравнений

Рассмотрим систему алгебраических уравнений

$$\begin{cases} x_1^2 - x_1 = 0 \\ \dots\dots\dots\dots \\ x_{n-2}^2 - x_{n-2} = 0 \\ x_{n-1}^2(n-2-x_1-\dots-x_{n-2}) + \quad x_{n-1} + 1 = 0 \\ x_1 + \dots + x_n - n + 2 = 0 \end{cases}$$

Эта система уравнений имеет единственное вещественное решение

$$\begin{cases} x_1 = 1 \\ \dots\dots\dots\dots \\ x_{n-2} = 1 \\ x_{n-1} = -1 \\ x_n = 1 \end{cases}$$

Это решение будет получено алгоритмом А, только в том случае, если для первых (n-2) уравнений оракул укажет в качестве решений именно решение

$$\begin{cases} x_1 = 1 \\ \dots\dots\dots. \\ x_{n-2} = 1 \end{cases}$$

Отметим, что общее число решений этой системы равно 2^{n-2}. Базис Гребнера относительно лексикографического порядка совпадает с левыми частями частями уравнений, т.е. имеет ту же сложность, что и описание идеала. В работе [2] был приведен пример идеала, для которого базис Гребнера с экспоненциального размера относительно описания самого идеала. В этом случае алгоритм А всегда приводит к рациональному (целочисленному) решению, поскольку иных решений попросту нет.

Литература.

[1]. Ван дер Варден Б.Л., Алгебра, Москва, Наука, 1976.

[2]. А.В. Шокуров, Сравнение сложностей задач нахождения базиса Гребнера идеала и решений этого идеала. Труды Института системного программирования РАН, том 22, 2012 г. ISSN 2220-6426 (Online), ISSN 2079-8156 (Print), стр.

[3]. Faugere J.C., Gianni P., Lazard D., Mora T., Efficient computation of zero-dimensional Grobner bases by change of ordering, Journal of Symbolic Computation, 1993,v.16, issue 4,pp.329-344.

On Solving The Systems of Algebraic Equations Using Gröbner Bases

Alexander Shokurov, shok@ispras.ru
ISP RAS

Abstract. Described and proved the algorithm for finding some solution of algebraic equations over arbitrary field k for zero dimension ideals if Gröbner basis of this ideal over lexicographic order is given. The found Solution lies in the algebraic closure of k. An example for a system of algebraic equations having a unique solution in the main field, and exponentially many solutions of this system is suggested.

Keywords. Gröbner basis, ideal,

References

[1]. B.L. van der Waerden, Algebra I, Achte Auflage der Modernen Algebra, Springer-Verlag Berlin New York 1971. Algebra II, Fünfte Auflage, Springer-Verlag Berlin New York 1967.

[2]. Shokurov A.V., Sravnenie slozhnostej zadach nakhozhdeniya bazisa Groybnera ideala i reshenij e'togo ideala [Comparing complexities of problems of determining of Grebner's basis of ideal and solving this ideal]. Trudy ISP RAN [The Proceedings of ISP RAS], 2012, vol. 22, pp. 463-474. (in Russian)

[3]. Faugere J.C., Gianni P., Lazard D., Mora T., Efficient computation of zero-dimensional Gröbner bases by change of ordering, Journal of Symbolic Computation, 1993,v.16, issue 4,pp.329-344.

Оптимальное упорядочение конфликтующих объектов и задача коммивояжера

А.В. Воеводин, С.А. Косяченко
Silver-AVV@yandex.ru , spiero@yandex.ru
Видео Интернешнл

Аннотация. В работе представлена постановка задачи оптимального упорядочения конфликтующих объектов и ее связь с задачей коммивояжера (Travelling Salesman Problem или TSP). Задача оптимального упорядочения конфликтующих объектов возникает в социологии, при анализе графов в социальных сетях, при размещении рекламных заказов в сетях СМИ. В статье описаны используемые авторами на практике быстрые алгоритмы решения этой и связанных с ней задач. Также рассмотрена задача TSP с разреженной матрицей штрафов. Для задач TSP с ленточной и блочно-диагональной матрицами найдены необходимые и достаточные условия и построены точные алгоритмы, при которых достигается нулевое минимальное значение целевой функции задачи. Предложены эффективные алгоритмы и для произвольных разреженных матриц. Приведены результаты аналитических и численных исследований сложности разработанных алгоритмов и точности решения, а также рекомендации по применению алгоритмов для решения задач подобного типа.

Ключевые слова: оптимальное размещение; задача коммивояжера; TSP; NP-трудные задачи; ленточные матрицы; разреженные матрицы; жадный алгоритм; штрафная функция; конфликты, сети СМИ.

Введение

Задача оптимального размещения конфликтующих объектов обобщает задачу коммивояжера, такие задачи возникают во многих предметных областях. Это классическая задача коммивояжера обхода всех городов по кратчайшему пути [1], задача о сверлильном станке, задача о производстве красок, пополнение банкоматов наличными деньгами, сбор сотрудников вахтовым методом, расклейка афиш, другие задачи логистики [2]. В общем случае в рассматриваемой нами задаче не все объекты связаны и конфликтуют между собой, а назначаемые за конфликт штрафы могут быть большими при непосредственном соседстве объектов и уменьшаются при отдалении их друг от друга в списке. Для задач коммивояжера обычно сушествует один тип

штрафа, например: расстояние, стоимость или время доставки груза между городами. В общем случае может быть несколько типов штрафов, определяемых разными причинами нежелательного соседства объектов, например, принадлежностью их к конфликтующим группам по различным признакам. Такие задачи возникают в социологии, при нахождении оптимальных путей между вершинами графа в социальных сетях, при размещении рекламных заказов в сетях СМИ. Несмотря на то, что данная задача в общем случае относится к классу NP-трудных задач, для наиболее распространенных частных случаев задачи TSP с разреженной матрицей штрафов построены точные быстрые алгоритмы. Для общего случая предложен эвристический быстрый алгоритм, масштабируемый до произвольного размера при сохранении линейной сложности при незначительных потерях в качестве решения. Это позволяет нам практически применять предложенные алгоритмы на больших объемах данных.

Постановка задачи и ее связь с задачей коммивояжера

Дано неупорядоченное множество из *N* объектов x_i, *i=1,..,N*. Требуется упорядочить эти объекты в виде линейного списка. Объекты «конфликтуют» между собой, конфликтующие объекты желательно разместить в списке подальше друг от друга. Математически конфликт выражается в том, что за непосредственное соседство в списке объектов x_i и x_j применяется штраф $p_{ij} \geq 0$. Если же между x_i и x_j в списке находится еще один объект x_k, то штраф за соседство x_i и x_j уменьшается в 2 раза. Если между x_i и x_j в списке находятся два других объекта, то штраф за соседство x_i и x_j уменьшается в 3 раза и т.д. В общем случае если после упорядочения в списке между объектами x_i и x_j оказалось $L_{ij} \geq 0$ других объектов, то за такую близость x_i и x_j применяется штраф, равный $\frac{p_{ij}}{L_{ij}+1}$. К примеру, при *N*=4 перестановке объектов x_3, x_1, x_4, x_2 будет соответствовать суммарный штраф

$$W = p_{31} + p_{14} + p_{42} + (p_{34} + p_{12})/2 + p_{32}/3.$$

Требуется среди всех перестановок объектов найти перестановку с минимальной величиной штрафа W. Формально постановка задачи выглядит следующим образом. Имеется множество из *N* объектов $x_1, x_2, \dots, x_N$. На исходной нумерации этих объектов задается матрица штрафов *P* с элементами $p_{ij} \geq 0$ за непосредственное соседство в списке объектов x_i и x_j. Пусть S – множество всех перестановок s_k чисел 1, 2,..., *N*, где *k=1,…, N!*. Обозначим $L_{ij}^k \geq 0$ - количество объектов из исходного множества, оказавшихся в перестановке s_k между объектами x_i и x_j. Необходимо среди всех перестановок s_k объектов найти перестановку с минимальным суммарным штрафом W_k:

$$\min_k W_k = \min_k \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{p_{ij}}{L_{ij}^k + 1} \qquad (1)$$

В частном случае, когда штрафуется только непосредственное соседство объектов, все значения $L_{ij}^k = 0$, и штраф p_{ij} применяется только N - 1 раз к непосредственно соседствующим объектам. В этом случае задачу (1) упорядочения конфликтующих объектов можно записать в виде

$$\min_k W_k^* = \min_k \sum_{i=1}^{N-1} p_{i,i+1} \qquad (2)$$

где минимум берется по всем *N!* перестановкам s_k индексов объектов x_i, $i = 1, 2,..., N$.

Оптимизационная задача (2) является классической задачей TSP, которая в теории сложности относится к классу NP-трудных задач. Для метрических задач TSP существуют приближенные алгоритмы полиномиальной сложности, которые гарантированно находят решение максимум вдвое отличающееся от оптимального[1]. Рассматриваемая нами задача (2) относится к варианту незамкнутой, симметричной, неметрической задачи TSP, для которых пока не предложено приближенных алгоритмов полиномиальной сложности с хорошей оценкой точности. Таким образом, в общем случае для задач (1) и (2) не приходится рассчитывать на существование полиномиально быстрых и точных алгоритмов решения. Однако для ряда распространенных на практике случаев нам удалось это сделать.

Заметим, что хотя задача (2) оптимального упорядочения конфликтующих объектов формально относится к задаче TSP, фактически она отличается от классической задачи коммивояжера нахождения оптимального пути обхода городов на плоскости и требует построения специальных алгоритмов. Даже метрическую задачу TSP не всегда можно изобразить на плоскости как задачу обхода городов. Например, рассмотрим задачу с N равноудаленными друг от друга вершинами, т.е. все $p_{ij} = C > 0, i \neq j,$ где $C -$ константа. Такое множество вершин можно изобразить на плоскости только при N=3. В общем случае N равноудаленных вершин потребуют не менее (N-1)-мерного пространства, где образуют (N-1)-мерный регулярный симплекс [3]. Неметрическая задача TSP, к которой относится рассматриваемая нами задача (2), еще менее похожа на классическую задачу коммивояжера. В ней не выполняется неравенство треугольника, ее можно представить только в виде взвешенного графа или в матричном виде.

Но самое большое отличие задачи оптимального упорядочения конфликтующих объектов (2) от классической задачи коммивояжера заключается в том, что не каждый объект конфликтует с каждым, т.е. в

матрице штрафов некоторые значения $p_{ij} = 0$. Более того, взвешенный граф задачи (2) может быть несвязанным и может распадаться на несвязанные друг с другом компоненты. Поскольку наша задача неметрическая, то стоимость «обходного» пути между вершинами i и j может быть малой или даже нулевой: $p_{i,k} + p_{kj} = 0$, а стоимость «непосредственного» пути по дуге p_{ij} большой. Когда в матрице штрафов много нулевых элементов p_{ij}, она называется разреженной. Но несмотря на все эти отличия от классической задачи коммивояжера задача упорядочения конфликтующих объектов (2) остается задачей TSP и даже может иметь смысловое наполнение близкое к классическому. Матрица штрафов может означать стоимость дополнительных, не входящих в базовый набор, услуг на пути обхода городов, например: стоимость проезда по платным дорогам, оплата номеров и парковки в отелях повышенного класса на трассе и т.п. Такая матрица может быть разреженной. В данной статье мы построим алгоритмы, эффективные для задач TSP с разреженной матрицей штрафов. Будут предложены процедуры сведения задачи с хаотически разреженной матрицей к задачам с ленточной или блочно-диагональной матрицей. В свою очередь для задач TSP с ленточной или блочно-диагональной матрицами будут найдены необходимые и достаточные условия и построены алгоритмы, при которых всегда существует путь с $W^* = 0$. Важные и часто встречающиеся на практике классы задач с ленточной и блочно-диагональной матрицами рассмотрены в работе отдельно.

Матричная постановка задачи

В матричном виде задача (2) сводится к одноименной перестановке строк и столбцов симметричной неотрицательной матрицы стоимости $P = (p_{ij})$, $p_{ij} \geq 0$, так, чтобы сумма чисел на 2-й диагонали, находящейся непосредственно над главной, была минимальна. Любая одноименная перестановка строк и столбцов матрицы эквивалентна преобразованию $T*P*T^t$, где T – матрица перестановок, T^t – транспонированная к ней матрица. Матрица перестановок – это 0-1 матрица, у которой в любой строке и любом столбце только один ненулевой элемент. Перемножение матриц перестановок снова дает матрицу перестановок. Матрица T^t также является матрицей перестановок. Умножение слева матрицы P на матрицу T приводит к перестановке строк матрицы P. Умножение справа матрицы P на матрицу T^t приводит к одноименной перестановке столбцов матрицы P. Матрица $T*P*T^t$ снова будет симметричной матрицей. В результате одноименных перестановок строк и столбцов матрицы P ее диагональные элементы будут оставаться на главной диагонали. Таким образом, матричная постановка задачи (2) следующая. Найти матрицу перестановок T такую, чтобы у матрицы $T*P*T^t$ сумма чисел на 2-й диагонали была минимальна.

Оптимальный алгоритм для задачи с блочно-диагональной матрицей

Определение. Матрица, которая перестановкой одноименных строк и столбцов может быть приведена к блочно-треугольному виду, называется разложимой.

В нашем случае симметричной матрицы разложимая матрица приводится очевидно к блочно-диагональному виду. Симметричная матрица P разложима тогда и только тогда, когда ее граф несвязен. Справедлива следующая

Лемма 1. Если матрица P порядка N разложима и максимальный размер блока в ее блочно-диагольном представлении $\leq \left\lceil \frac{N}{2} \right\rceil$, то минимум в задаче (2) оптимального упорядочения объектов равен 0.

Приведем алгоритм, доказывающий лемму 1 и обеспечивающий оптимальное решение задачи (2).

Алгоритм 1. Оптимальное упорядочение строится следующим образом. Сначала в списке поочередно располагаем все первые вершины из каждого блока матрицы, затем вторые вершины из каждого блока матрицы и т.д. Поскольку при таком алгоритме любые две рядом стоящие в списке вершины i и j будут принадлежать разным диагональным блокам матрицы, они не будут связаны между собой, т.е. для любых соседних вершин $p_{ij} = 0$, а значит значение W^* суммы ЦФ в задаче (2) равно 0. Алгоритм требует N-1 шагов.

Следствие. Если граф задачи (2) несвязен, и максимальное число вершин в связной компоненте графа $\leq \left\lceil \frac{N}{2} \right\rceil$, то минимум в задаче (2) оптимального упорядочения объектов равен 0.

Отметим, что если в условиях леммы 1 использовать попарные перестановки вершин, то потребуется $\leq \left\lceil \frac{N}{2} \right\rceil$ попарных перестановок. Перестановке вершин i и j соответствуют попарные одноименные перестановки i-й и j-й строк и i-го и j-го столбцов матрицы P. В любой попарной одноименной перестановке строк и столбцов среди интересующих нас элементов 2-й диагонали матрицы P меняются только 4 элемента на новые. Точнее при перестановке вершин i и j элементы меняются следующим образом:

$$p_{i-1,i} \leftrightarrow p_{i-1,j},\ p_{i,i+1} \leftrightarrow p_{i+1,j},\ p_{j-1,j} \leftrightarrow p_{i,j-1},\ p_{j,j+1} \leftrightarrow p_{i,j+1}. \qquad (3)$$

Здесь мы учитываем, что матрица P симметричная. Таким образом, при попарной перестановке достаточно в формуле суммы целевого функционала (2) подменить только 4 слагаемых. Если какой-либо из указанных индексов выходит из диапазона $\overline{1,N}$, то соответствующий элемент не переставляется и в сумме не участвует.

Алгоритм 1, обеспечивающий нулевой минимум в задаче (2) для разложимой матрицы и ее несвязанного графа, можно успешно применять в качестве приближенного алгоритма к матрицам близким к блочно-диагональным, у

которых все элементы, находящиеся вне диагональных блоков, малы относительно элементов внутри диагональных блоков.

Оптимальные алгоритмы для задачи с ленточной матрицей

Другим важным классом матриц, для которых можно построить алгоритм с нулевым значением штрафа в задаче (2), является класс ленточных симметричных матриц.

Определение. Матрица P порядка $N \geq 2$ называется ленточной, если

$$p_{ij} = 0, \qquad \text{при } |i-j| \geq l,$$

Величина $l > 0$ называется полушириной ленты такой матрицы.

Например, трехдиагональная матрица будет иметь полуширину ленты $l = 2$. Напомним, что в рассматриваемых нами задачах (1) и (2) матрица P симметричная.

Лемма 2. Если матрица P порядка N является ленточной с полушириной ленты $l \leq \left[\frac{N}{2}\right] - 1$, то минимум в задаче (2) оптимального упорядочения объектов равен 0.

Приведем алгоритм попарных перестановок строк и столбцов, обеспечивающий утверждение леммы 2.

Алгоритм 2. Для $N \leq 5$ утверждение очевидно, поскольку тогда $l \leq 1$ и матрица P имеет нулевую 2-ю диагональ. Пусть $N > 5$. Индексы i, j попарных одноименных перестановок строк и столбцов будем брать как индексы элементов матрицы P, лежащие на ее $\left(\left[\frac{N}{2}\right] + 1\right) -$ й диагонали. Брать следует индексы не всех элементов с этой диагонали, а через один, начиная с 1-го. Число таких элементов будет $\left[\frac{N}{4}\right]$. Попарные перестановки выполняем последовательно в соответствии с индексами указанных элементов, начиная с 1-го верхнего элемента данной диагонали. При перестановках вершин i и j используем формулы (3). Из них видно, что мы последовательно переставляем четверки соседних с p_{ij} нулевых элементов, находящихся вне ленты матрицы, на ее 2-ю диагональ.

Отметим, что при попарных одноименных перестановках строк и столбцов с индексами i, j ленточной матрицы происходит заполнение элементов матрицы в i-й строке и j-м столбце, отстоящих от p_{ij} на $\leq l - 1$ позиций. Но этот факт не мешает работе данного алгоритма, поскольку индексы с $\left(\left[\frac{N}{2}\right] + 1\right) -$ й диагонали матрицы P берутся через один. Алгоритм 2 требует $\left[\frac{N}{4}\right]$ попарных перестановок одноименных строк и столбцов матрицы.

Ограничение на полуширину ленты в лемме 2 можно ослабить, если использовать алгоритм с бóльшим числом перестановок. А именно, справедливо следующее утверждение.

Лемма 3. Если матрица P порядка N является ленточной с полушириной ленты $l \le \left[\frac{N}{2}\right]$, то минимум в задаче (2) оптимального упорядочения объектов равен 0.

Приведем алгоритм, обеспечивающий утверждение леммы 3.

Алгоритм 3. Оптимальное упорядочение строится следующим образом. Рассмотрим упорядочение вершин 1,2,… N, задаваемое исходной ленточной матрицей. В оптимальном списке располагаем эти вершины подряд в каждую четную позицию, начиная со 2-й позиции. Достигнув конца списка, вершины располагаем подряд в каждую нечетную позицию, начиная с 1-й позиции. Оптимальное упорядочение построено. Например, для $N = 8$ максимальная полуширина ленты матрицы P будет $l = 4$. Оптимальное упорядочение вершин будет таким: 5,1,6,2,7,3,8,4. Такое упорядочение обеспечивает нулевой минимум в задаче (2), ибо из структуры исходной ленточной матрицы следует, что в построенном оптимальном списке любые две рядом стоящие вершины i и j не связаны между собой, так как для них $p_{ij} = 0$. Так, в рассматриваемом примере в исходной ленточной матрице элементы $p_{51} = p_{16} = p_{62} = p_{27} = p_{73} = p_{38} = p_{84} = 0$. Значит, для указанного оптимального списка вершин 5,1,6,2,7,3,8,4 минимум в задаче (2) равен 0. Алгоритм 3 требует N-1 шагов – примерно в 4 раза больше, чем алгоритм 2.

Следствие. Если для графа задачи (2) существует упорядочение вершин, при котором любая вершина в списке связана только с вершинами, отстоящими от нее не более, чем на $\left[\frac{N}{2}\right] - 1$ позиций в списке, то минимум в задаче (2) оптимального упорядочения объектов равен 0.

Действительно, такое упорядочение вершин описывается ленточной матрицей с полушириной ленты $\le \left[\frac{N}{2}\right]$. Значит, в силу леммы 3, $W^* = 0$.

Отметим, что указанное в лемме 3 ограничение на полуширину ленты $\le \left[\frac{N}{2}\right]$ является предельным. Если полуширина ленты превышает это значение, то нулевое значение функционала задачи (2) гарантировать уже нельзя. Точнее справедлива следующая

Теорема 1. Пусть P - симметричная ленточная матрица порядка N, у которой все $p_{ij} > 0$ ($i \ne j$) в пределах ее ленты. Для того чтобы минимум в задаче (2) оптимального упорядочения объектов был равен 0, необходимо и достаточно, чтобы полуширина ее ленты $l \le \left[\frac{N}{2}\right]$.

Доказательство. Достаточность условия следует из леммы 3. Докажем необходимость от противного: если $l > \left[\frac{N}{2}\right]$, то $W^* > 0$. Пусть полуширина ленты $l \geq \left[\frac{N}{2}\right] + 1$. Тогда, поскольку ширина ленты L=2*l-1, получаем, что $L \geq 2 * \left[\frac{N}{2}\right] + 1 \geq N$. То есть ширина ленты $L = N$. А по условиям теоремы все $p_{ij} > 0$ ($i \neq j$) в пределах ленты. Значит в матрице P существует строка с некоторым номером k, в которой все ее элементы $p_{kj} > 0$ для $j \neq k$ от 1 до N, и столбец с некоторым номером k, в котором все его элементы $p_{ik} > 0$ для $i \neq k$ от 1 до N. Другими словами вершина с номером k конфликтует со всеми другими вершинами. А поскольку при любых перестановках вершина с номером k будет соседствовать хотя бы с одной другой вершиной, то для фунционала в задаче (2) будет справедлива оценка

$$W^* \geq \min_{i \neq k} p_{ik} > 0,$$

что и требовалось доказать.

Алгоритмы 2 и 3, обеспечивающие нулевой минимум в задаче (2), можно успешно применять в качестве приближенных алгоритмов к матрицам близким к ленточным, у которой все элементы, находящиеся вне ее ленты, малы относительно элементов внутри ленты. Для приведения матрицы к ленточной или близкой к ленточной можно применять специальные алгоритмы.

Алгоритмы для задач с разреженными и матрицами общего вида

Для случая разреженных матриц общего вида, когда в матрице P ненулевые элементы расположены хаотично, можно использовать эффективные методы группирования ненулевых элементов на диагоналях близких к главной. Например, это прямой и обратный методы Катхилл-Макки для разреженных матриц, которые с помощью одноименных перестановок строк и столбцов приводят матрицу к ленточному виду с шириной ленты значительно меньшей, чем у исходной. Сложность этой процедуры $O(N^2)$ [4]. В работах [4,5] приводятся многочисленные примеры, когда, применяя прямой или обратный методы Катхилл-Макки к разреженной матрице размера N, получаем ленточную матрицу с шириной ленты в несколько раз или даже на порядок меньше, чем у исходной матрицы. А согласно теореме 1 для достижения нулевого минимума функционала достаточно всего лишь, чтобы полуширина ленты была $\leq \left[\frac{N}{2}\right]$. В этом случае, после процедуры Катхилл-Макки следует применить алгоритмы 2 или 3 оптимального упорядочения для достижения нулевого минимума функционала задачи (2).

Этот же подход можно успешно применять и в случаях, когда матрица P не является разреженной, то есть P – матрица общего вида, но в ней явно выделяются большие и малые элементы. Для матриц такого вида также сначала используем методы Катхилл-Макки для группирования больших элементов на диагоналях близких к главной. Затем применяем алгоритмы 2 или 3 оптимального упорядочения для минимизации функционала задачи (2). Таким образом мы построили новый быстрый алгоритм приближенного решения классической общей задачи TSP. Этот алгоритм будет достаточно точно работать в тех случаях, когда элементы p_{ij} в матрице штрафов P достаточно сильно отличаются по величине друг от друга, а число «малых» элементов в матрице не меньше $N^2/4$.

Алгоритмы для задачи с матрицей общего вида

Несмотря на существование оптимальных быстрых алгоритмов для блочно-диагональных, ленточных и разреженных матриц, а также эффективных алгоритмов для близким к ним матриц, в общем случае в задачах (1) и (2) не приходится рассчитывать на достаточно точные алгоритмы полиномиальной сложности. Сложность алгоритма полного перебора составит *O(N!)*. Если для малых N выполнение полного перебора возможно, то с ростом N мы столкнёмся с проблемой нехватки вычислительных ресурсов. Ограничения в вычислительных ресурсах приводят к необходимости построения экономичного в вычислительном смысле алгоритма поиска решения, близкого к оптимальному.

Применим к решению задач “жадный” алгоритм [6] (greedy algorithm), в котором каждый шаг является локально оптимальным. Рассмотрим работу жадного алгоритма для поставленной выше задачи (1) размещения на примере.

Алгоритм 4. Пусть есть множество объектов (x_1, x_2, x_3, x_4). Матрица попарных штрафов имеет вид: $P = \begin{vmatrix} 0 & 1 & 2 & 2 \\ 1 & 0 & 3 & 3 \\ 2 & 3 & 0 & 1 \\ 2 & 3 & 1 & 0 \end{vmatrix}$.

Так как в работе жадного алгоритма нам необходимо начинать размещение с самых конфликтных объектов, упорядочим объекты в порядке убывания «конфликтности», которую определим как сумму элементов соответствующей строки матрицы P. Получим вектор конфликтности $\bar{p}$=(5,7,6,6). В соответствии с ним будем размещать объекты в порядке (x_2, x_3, x_4, x_1).

Ищем оптимальное размещение для x_2, ставим x_2 в первую доступную позицию:

x_2			

С учетом уже поставленного x_2 ищем оптимальное размещение для x_3:

x_2			x_3

С учетом уже поставленных x_2 и x_3 ищем оптимальное размещение для x_4:

x_2	x_4		x_3

Наконец, ставим x_1 в последнюю доступную позицию:

x_2	x_4	x_1	x_3

Получаем штраф W=9.

Основным недостатком «жадного» алгоритма является оптимизация только одного следующего шага. Для его улучшения авторы предлагают воспользоваться следующим эвристическим алгоритмом.

Алгоритм 5.

1. Если $N \leq 7$, то выполняем оптимальное размещение полным перебором и на выход. Иначе переходим к шагу 2.
2. Все объекты сортируются в порядке убывания «конфликтности». Пронумеруем их заново $(x_1, x_2, \dots, x_N)$. Смещение $S = 0$. Список размещения – пустой.
3. Берем 7 объектов $(x_{S+1}, \dots, x_{S+7})$ и с помощью полного перебора ищем оптимальное размещение этого подмножества в списке.
4. Если $S+7=N$, то все объекты размещены - Выход.
5. Из всех размещенных 7 объектов оставляем в списке только первый x_{s+1}, остальные убираем.
6. $S = S + 1$. Возвращаемся к шагу 2.

Отличием алгоритма 5 от приведенного выше жадного алгоритма 4 заключается в том, что оптимальное размещение x_{S+1} ищется не локально оптимально, а с учетом других конфликтующих объектов, что позволяет лучше определять оптимальную позицию для x_{S+1}.

Если алгоритм 5 применить к вышеприведенному примеру, то выполняя полный перебор даже для 2 объектов вместо 7, получим оптимальное значение суммарного штрафа W=7.5.

В данном случае нам удалось добиться глобального минимума, хотя понятно, что в общем случае данный алгоритм на такой результат не претендует.

Как показывает практика, предложенный алгоритм 5 удовлетворительно решает задачи размером $N \leq 30$. При N>30 скорость работы алгоритма начинает заметно падать. Для компенсации этого недостатка для больших N

предлагается разбивать позиции списка на подблоки таким образом, чтобы длина каждого подблока была не более 30, и проводить оптимизацию в каждом подблоке «почти» отдельно.

Алгоритм 6. Имеем N>30 объектов, они отсортированы в порядке убывания штрафа $(x_1, x_2, \dots, x_N)$. Вычисляем количество подблоков $K=\left\lceil\frac{N}{30}\right\rceil$. Начиная с первого объекта, распределяем последовательно все объекты $(x_1, \dots, x_N)$ поочередно по подблокам 1,...,K. Когда подблоки кончатся, распределяем объекты с последнего подблока до первого, затем снова с первого до последнего и т.д. Таким образом, в каждом подблоке мы получим примерно одинаковый по конфликтности набор объектов. После такого разбиения предложенный выше алгоритм размещения применяем к каждому подблоку отдельно. А для того, чтобы избежать размещения конфликтующих объектов на границе подблоков будем применять приведенный выше алгоритм еще и на границе подблоков следующим образом.

Рассмотрим пример. Пусть имеется 2 подблока. Все множество объектов соответственно поделено на 2 группы объектов, которые должны быть размещены в подблоках 1 и 2 соответственно. Сначала выполняется размещение в 1-м подблоке. Затем размещаем объекты 2-ого подблока с учетом размещения 1-ого подблока. Затем выделяем промежуточный подблок на границе 1-ого и 2-ого подблока. В данный промежуточный подблок входят $\left\lceil\frac{l1}{2}\right\rceil$ последних объектов 1-ого подблока и $\left\lceil\frac{l2}{2}\right\rceil$ первых объектов 2-ого подблока, где $l1$, $l2$ – количество объектов в 1-ом и во 2-ом подблоке соответственно. Выполняем размещение в промежуточном подблоке с учетом расстановки первых $l1-\left\lceil\frac{l1}{2}\right\rceil$ объектов 1-ого подблока. Далее еще раз проводим переразмещение объектов во 2-ом подблоке. Таким образом, удается улучшить размещение на границе подблоков. Заметим, что результат работы предложенного алгоритма 6 может быть далек от оптимального, поэтому для конечной «шлифовки» будем дополнительно использовать алгоритм локальной оптимизации, приведенный ниже.

Алгоритм локальной оптимизации с помощью попарных перестановок

Будем менять местами все пары вершин (x_i, x_j), если это приводит к уменьшению ЦФ. Так поступаем до тех пор, пока происходит уменьшение ЦФ. Можно также на каждом шаге выбирать наилучшую перестановку вершин i и j, у которой (сумма старых 4 элементов 2-й диагонали) - (сумма новых 4 элементов 2-й диагонали) максимальна. При этом учитываем, что в любой попарной одноименной перестановке строк и столбцов среди интересующих нас элементов 2-й диагонали матрицы P меняются только 4 старых элемента на новые. Индексы старых и новых элементов определяются формулами (3). Так поступаем, пока наилучшая перестановка будет давать

выигрыш по функционалу. Такой метод сходится к локальному минимуму за конечное число шагов. Сложность одного шага $O(N^2)$. Если такую процедуру попарной перестановки применять только внутри каждого подблока, то общая сложность всех попарных перестановок и предложенного алгоритма упорядочения всех конфликтующих объектов в целом будет равна $O(N)$.

Общая схема функционирования предложенного алгоритма приведена на рисунке 1.

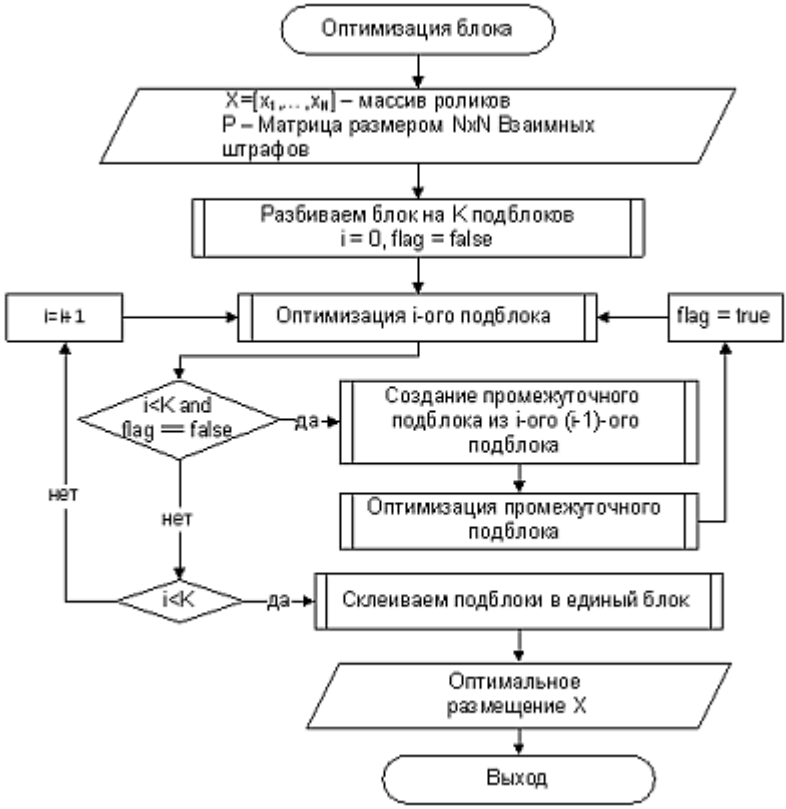


Рис. 1.

Сложность предложенного алгоритма в целом линейна и равна $C*N/30$, где C – число операций, приходящихся на обработку одного подблока размера не более 30.

Численные исследования

Оценим численно точность и скорость работы предложенного выше алгоритма. Определить точность работы можно, зная оптимальное значение ЦФ, которое можно получить алгоритмом полного перебора. Ввиду большой ресурсоемкости такого подхода сделать это представляется возможным только для небольших N. Результаты сравнительных исследований для $N \leq 10$ приведены в таблице 1. В колонке «Объекты» в таблице 1 приведена кодировка набора объектов, на котором проводился замер точности работы алгоритмов. Например, кодировка «1112223» обозначает, что имеется 7 объектов, которые поделены на 3 подгруппы: три одинаковых объекта «1», три одинаковых объекта «2» и объект «3». Каждый объект конфликтует только с объектами своей подгруппы с одинаковым единичным штрафом. Так как решение задачи (2) нахождения минимума W^* тривиально, в данной задаче мы искали только минимум W в задаче (1).

Таблица 1. Значения ЦФ для размещения 10 и менее объектов.

	Объекты	Алгоритм полного перебора. Значение ЦФ $W_{\text{опт}}$	Предложенный алгоритм. Значение ЦФ $W_{\text{алг}}$.
7	1112223	2.58	2.58
8	11122334	2.26	2.26
9	111122233	3.82	4.02
10	1111222334	3.35	3.52

При больших значениях N нет возможности найти оптимальную расстановку с помощью алгоритма полного перебора, но можно провести специальные тесты с очевидным решением.

Тест 1. Возьмем множество из 100 объектов, которое состоит из двух подмножеств: $(x_1, x_3, x_5 \ \ldots x_{99})$, $(x_2, x_4, x_6 \ \ldots x_{100})$, по 50 одинаковых объектов в каждом. Каждый объект конфликтует только с объектами своего подмножества с одинаковым единичным штрафом. Будем искать минимум W в задаче (1). Оптимальная расстановка для данного теста совпадает с порядком следования объектов $x_1, \ldots x_{100}$ или, другими словами, оптимальным размещением будет являться чередование объектов из первого подмножества с объектами из второго подмножества. Подсчитаем значение $W_{\text{опт}}$ для задачи (1):

$$\boldsymbol{W}_{\text{опт}}(\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}\ \ldots \boldsymbol{x_{100}}) = \boldsymbol{2} \cdot \boldsymbol{W}_{\text{опт}}(\boldsymbol{x_1}, \boldsymbol{x_3}, \boldsymbol{x_5}\ \ldots \boldsymbol{x_{99}}) =$$

$$\boldsymbol{2} \cdot \begin{pmatrix} \frac{\boldsymbol{1}}{\boldsymbol{2}} + \frac{\boldsymbol{1}}{\boldsymbol{4}} + \frac{\boldsymbol{1}}{\boldsymbol{6}} + \cdots + \frac{\boldsymbol{1}}{\boldsymbol{98}} + \\ \frac{\boldsymbol{1}}{\boldsymbol{2}} + \cdots + \frac{\boldsymbol{1}}{\boldsymbol{96}} + \\ + \cdots + \\ + \frac{\boldsymbol{1}}{\boldsymbol{2}} \end{pmatrix} \quad (4)$$

Здесь в первой строке записан штраф за конфликты объекта x_1 и всех остальных объектов первого подмножества $(x_3, x_5\ \ldots x_{99})$. Во второй строке штраф за конфликты x_3 и всех остальных объектов первого подмножества, исключая x_1, и т.д. После преобразований формул (4), получим оптимальное значение функции штрафа

$$W_{\text{опт}} = 2 \cdot \sum_{i=1}^{49} \frac{(50 - i)}{2 \cdot i} = 174.96$$

Применяя к тому же входному множеству объектов предложенный алгоритм, получим $W_{\text{алг}}$ =175.86.

Тест 2. Возьмем набор из 100 объектов $x_1, x_2, \ldots x_{100}$. Матрицу попарных штрафов определим следующим образом:

$$\begin{cases} p_{ij} = 1, & \frac{|i-j|}{5} \in Z \\ p_{ij} = 0, & \frac{|i-j|}{5} \notin Z \end{cases}$$

Другими словами, из первоначального множества X конфликтуют между собой только элементы из 5-ти подмножеств: $(x_1, x_6, x_{11}\ \ldots x_{96})$, $(x_2, \ldots x_{97})$, $(x_3, \ldots x_{98})$, $(x_4, \ldots x_{99})$, $(x_5, \ldots x_{100})$. Оптимальная расстановка для данного теста совпадает с порядком следования объектов $x_1, \ldots x_{100}$.

Посчитаем оптимальный штраф для данного размещения. Ввиду того, что конфликтующие подмножества одинаковы с точки зрения штрафа, мы можем подсчитать штраф для одной группы, например $(x_1, \ldots x_{96})$, и умножить его на 5. Воспользовавшись определением ЦФ (1), получаем:

$$\boldsymbol{W}_{\text{опт}}(\boldsymbol{x_1, x_2, \dots x_{100}}). = \mathbf{5} \cdot \boldsymbol{W}_{\text{опт}}(\boldsymbol{x_1, x_6 \dots x_{96}}) =$$

$$\mathbf{5} \cdot \begin{pmatrix} \frac{1}{5} + \frac{1}{10} + \frac{1}{15} + \cdots + \frac{1}{95} + \\ \frac{1}{5} + \cdots + \frac{1}{90} + \\ + \cdots + \\ + \frac{1}{5} \end{pmatrix} \qquad (4)$$

Здесь в первой строке записан штраф за конфликты x_1 и всех остальных объектов первого подмножества $(x_1, x_6, x_{11} \dots x_{96})$. Во второй строке штраф за конфликты x_6 и всех остальных объектов первого подмножества, исключая x_1, и т.д. После преобразований (4) получаем:

$$W_{\text{опт}} = 5 \cdot \sum_{i=1}^{19} \frac{(20 - i)}{5 \cdot i} = 51{,}95$$

Применим к первоначальному подмножеству предложенный алгоритм, получаем $W_{\text{алг}}$ =52,76.

Тест 3. Возьмем 34 объекта, которые поделены на 2 неконфликтующие группы: 12 объектов первой группы конфликтуют между собой с штрафом p=10 и 22 объекта 2-ой группы с штрафом p=2. Оптимальное размещение в этом случае будет иметь вид:

x_1	x_2	x_2	x_1	x_2	x_2	x_1	…		x_2	x_2	x_1

где x_1- объект из 1-ой группы, x_2 - объект из 2-ой группы.

Оптимальное значение ЦФ для такого размещения: $W_{\text{опт}}$=186,12.

Предложенный алгоритм находит решение: $W_{\text{алг}}$ =188,0.

Тест 4. Проведем тест аналогичный тесту 2, только с 3-мя группами по 30 взаимно не конфликтующих объектов. Т.е. имеем объекты $x_1, x_2, \dots x_{90}$. Функция попарных штрафов:

$$\begin{cases} p_{ij} = 1, & \frac{|i-j|}{3} \in Z \\ p_{ij} = 0, & \frac{|i-j|}{3} \notin Z \end{cases}$$

Оптимальная расстановка для данного теста совпадает с порядком следования объектов $x_1, \dots x_{90}$.

$$\boldsymbol{W_{\text{опт}} = 3 \cdot W_{\text{опт}}(x_1, x_4 \dots x_{87}) = 3 \cdot \begin{pmatrix} \frac{1}{3} + \frac{1}{6} + \frac{1}{9} + \dots + \frac{1}{87} + \\ \frac{1}{3} + \dots + \frac{1}{84} + \\ + \dots + \\ + \frac{1}{3} \end{pmatrix}} \quad (4)$$

После преобразований получаем:

$$W_{\text{опт}} = 3 \cdot \sum_{i=1}^{29} \frac{(30 - i)}{3 \cdot i} = 89{,}85$$

Применим к первоначальному подмножеству предложенный алгоритм, получаем $W_{\text{алг}}$ =90,38.

При подготовке данной статьи авторы пытались найти худшие примеры, в которых значение ЦФ предложенного алгоритма значительно отличалось бы от оптимального значения ЦФ, но это оказалось не так просто. Результаты численных исследований решения оптимизационной задачи (1) для ЦФ W и задачи (2) для ЦФ W^* приведены в таблице .

Табл. 2. Сводная таблица результатов численных исследований.

№ примера	$W_{\text{опт}}$	$W_{\text{алг}}$	$\frac{W_{\text{алг}} - W_{\text{опт}}}{W_{\text{алг}}}$ *100	$W^*_{\text{опт}}$	$W^*_{\text{алг}}$
1	174.96	175.86	0,51	0,0	3,0
2	51,95	52,76	1,54	0,0	0,0
3	186,12	188,0	1,00	18,0	18,00
4	89,85	90,38	0,59	0,0	0,0

Скорость работы предлагаемого алгоритма в секундах для больших N приведена на графике, изображенном на рис. 2.

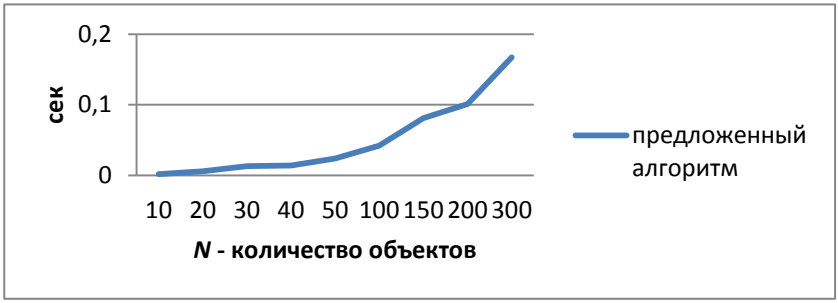


Рис. 1. Зависимость времени работы предложенного алгоритма от количества объектов

Сложность предложенного алгоритма упорядочения всех конфликтующих объектов в целом линейна и равна *O(N)*.

Заключение

В статье построены алгоритмы решения задач оптимального упорядочения конфликтующих объектов. Рассмотрены задачи двух типов: когда штрафуется только непосредственное соседство объектов (задача TSP), и обобщенный случай, когда штраф действует как на соседние, так и на отдаленные друг от друга конфликтующие объекты, и убывает в зависимости от числа расположенных между ними объектов. Показана связь задачи оптимального упорядочения конфликтующих объектов с задачей коммивояжера. Рассмотрена задача TSP с разреженной матрицей штрафов. Для наиболее распространенных случаев такой задачи TSP, когда матрица штрафов имеет вид ленточной или блочно-диагональной, построены точные быстрые алгоритмы, достигающие минимального нулевого значения целевой функции. Доказано необходимое и достаточное условие, при котором достигается нулевой минимум ЦФ в таких задачах TSP. Предложенные алгоритмы эффективны для разреженных матриц. Их также можно успешно применять в качестве приближенных алгоритмов к матрицам близким к ленточным, блочно-диагональным и разреженным. Для общего случая в статье предложен эвристический алгоритм, который показал высокое быстродействие при незначительных потерях в качестве решения. Для задачи TSP (2) он также работает эффективно. Удалось добиться хорошей его масштабируемости до произвольного размера при сохранении линейной сложности, что позволило нам использовать предложенные алгоритмы на практике на больших объемах данных, например, для задач размещения рекламных заказов в сетях СМИ. Их можно применять также для решения задач коммивояжера и близких задач обхода графов, в том числе в социологии, при нахождении оптимальных путей в социальных сетях.

Список литературы

[1]. Кузюрин Н.Н., Фомин С.А. Эффективные алгоритмы и сложность вычислений: Учебное пособие. – М.: МФТИ, 2007. - 312 с.

[2]. http://en.wikipedia.org/wiki/Travelling_salesman_problem

[3]. Peter Komjach: http://mathoverflow.net/questions/30270/maximum-number-of-mutually-equidistant-points-in-an-n-dimensional-euclidean-spac , geometry - Maximum number of mutually equidistant points in an n-dimensional Euclidean space is (n+1)_ Proof – MathOverflow, answered Jul 2, 2010.

[4]. Джордж А., Лю Дж. Численное решение больших разреженных систем уравнений: Пер. с англ. – М.: Мир, 1984. – 333 с.

[5]. Писсанецки С. Технология разреженных матриц: Пер. с англ. – М.: Мир, 1988. – 410 с.

[6]. Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. Алгоритмы: построение и анализ: Под ред. Красикова И. В. — 2-е изд. — М.: Вильямс, 2005. — 1296 с.

Optimal Ordering of Conflicting Objects and the Traveling Salesman Problem

Alexey Voevodin, Semen Kosyachenko
Silver-AVV@yandex.ru , spiero@yandex.ru
Business Center «Video International», Moscow, Russia

Abstract. The paper presents the setting of the problem of optimal ordering of conflicting objects related to the Travelling Salesman Problem. The problem of optimal ordering of conflicting objects appears in sociology, in graph analysis and finding in them optimal paths, in advertising in various media. Solution algorithms are described for this and related problems. The Travelling Salesman Problem with sparse matrix is also considered. For sparse practice cases of the Travelling Salesman Problem necessary and sufficient conditions are proved to objective function attained its minimum and the algorithms guaranteeing exact solution are constructed. The practical results of analytical and numerical investigations of algorithm complexity and solution accuracy are presented as well as recommendations for the algorithm applications to the solution of these problems.

Keywords: optimal placement; Travelling Salesman Problem; TSP; NP-hard problems; band matrix; sparse matrix; greedy algorithm; penalty function; conflicts, media network; advertising.

References

[1]. Kuzyurin N.N., Fomin S.A. Effektivnye algoritmy i slozhnost' vychislenij [Efficient algorithms and computational complexity]. Uchebnoe posobie [Tutorial]. Moscow, MIPT' Publ., 2007. 312 c.
[2]. http://en.wikipedia.org/wiki/Travelling_salesman_problem
[3]. Peter Komjach: http://mathoverflow.net/questions/30270/maximum-number-of-mutually-equidistant-points-in-an-n-dimensional-euclidean-spac, geometry - Maximum number of mutually equidistant points in an n-dimensional Euclidean space is (n+1)_ Proof – MathOverflow, answered Jul 2, 2010.
[4]. Alan George, Joseph W. H. Liu. Computer Solution of Large Sparse Positive Definite Systems. Prentice-Hall, Inc., Englewood Cliffs, N.J. 1981 – 324 pages.
[5]. Sergio Pissanetzky. Sparse Matrix Technology. Academic Press, 1984 – 321 pages.
[6]. Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford. Introduction to Algorithms (3rd ed.). MIT Press and McGraw-Hill. 2009 [1990].