Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем

Кафедра системного программирования

Власова Анна Сергеевна

Реализация алгоритма поиска путей с контекстно-свободными ограничениями для графовой базы данных Neo4j

Выпускная квалификационная работа бакалавра

Научный руководитель:
доцент, к. ф.-м. н. С. В. Григорьев

Рецензент:
программист ООО "ИнтеллиДжей Лабс" Р. Ш. Азимов

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and Administration of Information Systems

Software Engineering

Vlasova Anna

Implementation of Context-Free Path Querying for the Neo4j graph database

Bachelor's Thesis

Scientific supervisor:
Assistant Professor Semyon Grigorev

Reviewer:
software engineer IntelliJ Labs Co. Ltd. Rustam Azimov

Saint-Petersburg
2020

Оглавление

Введение

Графовые базы данных — базы данных, в которых семантически данные представляются в виде набора вершин и ориентированных ребер, а также их свойств. В случае, когда между хранящимися объектами имеются взаимосвязи, графовые базы данных могут давать существенное улучшение производительности запросов. В частности, в связи с этим они находят свое применение во многих областях [14]. Например, с помощью них можно моделировать социальные сети [4], граф потока управления программы [13], геномную информацию [8] или библиометрические данные (статьи — вершины, отношение цитирования — ребра). Для анализа таких представлений используются запросы к графовым базам данных. Запросы могут формулироваться как задача поиска путей, удовлетворяющих заданным условиям. Один из подходов для представления этих условий состоит в задании формального языка. Считается, что путь принадлежит формальному языку, если этому языку принадлежат сконкатенированные метки ребер данного пути [17].

Одной из известных графовых баз данных является Neo4j. Для написания запросов к ней используется декларативный язык Cypher [11]. Данный язык поддерживает только ограничения на пути, заданные в терминах регулярных языков. Существует более широкий класс языков — контекстно-свободные. Например, задачу поиска всех потомков одного поколения, задаваемую контекстно-свободной грамматикой $S \to aSb \mid \varepsilon$, не выразить в терминах регулярных выражений [15]. Отсюда возникает необходимость использования дополнительных средств для выполнения запросов с контекстно-свободными ограничениями.

Несмотря на то, что для решения задачи поиска путей, удовлетворяющих ограничениям в терминах формальной грамматики, уже существуют различные алгоритмы [10, 7, 3], все еще присутствует проблема их низкой производительности [5]. Поэтому возникает необходимость в создании новых алгоритмов и реализаций, решающих данную задачу. Как отмечается в статье [7], классические алгоритмы синтаксического

анализа хорошо адаптируются и обобщаются на графы. Одним из таких классических алгоритмов является Generalized LL [16]. Он поддерживает любые контекстно-свободные грамматики. Кроме того, у данного алгоритма имеются эффективные реализации с использованием различных оптимизаций. Поэтому в основе данной работы лежит идея о том, что можно подобрать производительную реализацию эффективного алгоритма синтаксического анализа, принимающего по умолчанию строку, и адаптировать ее с линейного входа на графы.

Подобная работа по обобщению со строк на графы уже была проделана на основе парсер-комбинаторов. Это решение ранее было интегрировано с Neo4j [12] и показало хорошие результаты. Оно является достаточно гибким, однако, возможно, что без использования парсер-комбинаторов удастся повысить скорость выполнения запросов.

1. Обзор

В данном разделе определены основные необходимые понятия из теории формальных языков и грамматик. Кроме того, определена задача поиска путей в графе с контекстно-свободными ограничениями и представлены существующие подходы для ее решения. Также описаны использующиеся в работе алгоритмы синтаксического разбора. Далее показывается, как подобный алгоритм может быть обобщен для решения задачи поиска путей с контекстно-свободными ограничениями. Представлена эффективная реализация выбранного алгоритма синтаксического разбора. В заключении этого раздела обоснован выбор базы данных, описаны ее основные преимущества.

1.1. Основные определения из теории формальных языков

Для начала определим ключевые понятия теории формальных языков: грамматику и язык, который она задает.

Определение 1 *Формальной грамматикой называется четверка* $\langle N, \Sigma, P, S \rangle$, *где*

- N — *набор нетерминальных символов;*
- Σ — *набор терминальных символов, не пересекающийся с* N*;*
- P — *набор допустимых правил (продукций), имеющих вид* $A \to B$, *где* $A \in \{N \cup \Sigma\}^+ \setminus \{\Sigma\}^+$ *и* $B \in \{N \cup \Sigma\}^*$*;*
- $S \in N$ — *стартовый нетерминал.*

Определение 2 *Множество всех выводимых из стартового нетерминала слов (цепочек терминалов) является языком, который задает грамматика. Выводом называется последовательное применение продукций грамматики.*

Поскольку в данной работе речь идет о поиске путей с контекстно-свободными ограничениями, необходимо определить соответствующую грамматику.

Определение 3 *Грамматику называют контекстно-свободной, если в левой части у всех продукций стоит единственный нетерминал.*

1.2. Поиск путей в графе с контекстно-свободными ограничениями

Определим формально задачу, которая рассматривается в данной работе. Пусть даны:

- контекстно-свободная грамматика $\mathbb{G} = \langle N, \Sigma, P, S \rangle$;
- ориентированный граф $G = \langle V, E, T \rangle$, где V — множество вершин графа, $E \subseteq V \times T \times V$ — множество его ребер, а $T \subseteq \Sigma$ — множество меток на ребрах, причем каждая метка является терминальным символом грамматики $\mathbb{G}$;
- множество стартовых вершин $V_S \subseteq V$ и финальных вершин $V_F \subseteq V$.

Язык, порожденный грамматикой $\mathbb{G}$ обозначим как $L(\mathbb{G})$. Рассмотрим путь в графе G:

$$p = (v_0, e_0, v_1, e_1, \cdots, e_{n-1}, v_n),$$

где $v_k \in V$ и $e_k = (v_{k-1}, t_k, v_k)$. Пути в графе сопоставим слово $T(p) = t_0 t_1 \cdots t_{n-1}$ — конкатенацию меток на ребрах данного пути. Тогда задача поиска путей в графе с контекстно-свободными ограничениями состоит в том, чтобы найти все такие пути в графе, что $T(p) \in L(\mathbb{G})$ и $v_0 \in V_S$, $v_n \in V_F$.

1.3. Существующие решения задачи КС запросов

В области запросов с контекстно-свободными ограничениями было проведено немало теоретических исследований, но, как правило, в них

не осуществлялось интеграции с базами данных и не проводились замеры производительности на реальных графах. Недавно интерес к задаче контекстно-свободных запросов возрос: авторы статьи [5] рассмотрели популярные алгоритмы в этой области и замерили их производительность с использованием графовой базы данных Neo4j. По результатам исследования авторы пришли к выводу, что на данный момент на больших графах решения оказываются не применимы из-за большого времени выполнения запросов. Это является мотивацией для поиска новых решений.

Одним из классов решений описанной задачи являются алгоритмы, основанные на матричных операциях [2]. Они отличаются высокой производительностью. Данные алгоритмы не возвращают сами пути, а лишь проверяют их наличие между всеми парами вершин.

Также есть решение, которое уже было интегрировано с Neo4j — библиотека Meerkat.Graph[1]. Она была реализована на Scala и представлена в статье [12]. Эта реализация основана на парсер-комбинаторах и поддерживает любые контекстно-свободные грамматики.

Помимо этого, исследования показывают, что базовые алгоритмы синтаксического анализа, принимающие строку и грамматику, хорошо обобщаются на графы, например, есть работы в которых описывается, каким образом можно решить задачу для графов на основе обобщенного LL-анализатора [7] или обобщенного LR-анализатора [6]. Один из этих подходов и рассматривается в данной работе.

1.4. Обобщенный LL-анализатор (GLL)

Классическим алгоритмом синтаксического разбора является LL-анализатор. Он реализуется с помощью нисходящего разбора выражения. Применим данный алгоритм только к LL-грамматикам, однако не все контекстно-свободные грамматики принадлежат классу LL(k) грамматик для какого-либо k. Также классический подход не предусматривает неоднозначностей и левой рекурсии в грамматике. В статье [16]

[1]Репозиторий библиотеки Meerkat.Graph: `https://github.com/YaccConstructor/Meerkat`. Дата обращения: 29.05.2020

было предложено его обобщение Generalized LL (GLL).

Стандартный рекурсивный спуск для каждого из нетерминалов предполагает наличие функции обработки. В случае не LL(k) грамматики может быть невозможно определить какую из функций надо вызвать, поэтому для обработки подобных ситуаций в GLL используется очередь, в которой хранятся состояния, описывающие каждый возможный выбор. Если в рекурсивном спуске произойдет ошибка при синтаксическом анализе внутри какой-либо из функций, это будет автоматически означать, что входная строка не принадлежит формальному языку. В случае GLL это будет означать, что необходимо рассмотреть альтернативное состояние — следующее, хранящееся в очереди.

Текущее состояние представляется дескриптором, который состоит из четырех составляющих:

- слот, то есть правило вида $N \to \alpha \ . \ x\beta$ (точка показывает текущую позицию в грамматике);
- стек;
- текущий индекс во входной строке;
- сжатое представление леса разбора (SPPF) — компактное представление множества деревьев вывода.

Для эффективного хранения множества различных стеков используется Graph Structured Stack (GSS) [18]. Благодаря использованию данной структуры объем требуемой памяти для хранения стеков не разрастается экспоненциально — и вычислительная, и пространственная сложности исходного GLL-парсера являются кубическими от размера входа.

1.5. Модификация GLL для графов

Описанный алгоритм может быть обобщен с разбора строк на графы так, как это было представлено в работе [7]. Пусть дано множество стартовых вершин V_S, множество конечных вершин V_F и грамматика G. Строку, которую принимает исходный алгоритм, можно рассматривать

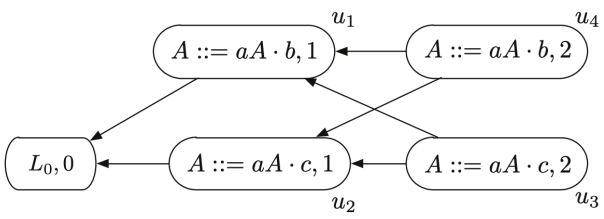
как линейный граф. Для корректного обобщения на графы необходимо сделать следующие модификации:

- исходное множество дескрипторов должно включать в себя дескрипторы для всех вершин из V_S;
- на том шаге, на котором берется следующий символ, теперь может быть несколько вариантов следующих символов (соответствующих меткам на всех ребрах, выходящих из текущей вершины);
- в случае, когда синтаксический разбор строки завершен, в оригинальном GLL осуществлялась проверка на то, что текущий индекс совпадает с концом строки . Теперь надо осуществлять проверку на принадлежность конечной вершины множеству V_F .

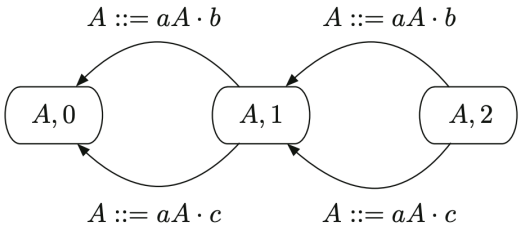
В случае графов сжатое представление леса разбора удобно использовать для получения всех найденных путей, при этом оно остается конечным даже в случае бесконечного количества путей.

1.6. Библиотека Iguana

В статье [1] описывается производительная реализация GLL. В ней используется эффективная модификация стека (GSS), в которой в GSS вершине вместо текущей позиции в грамматике хранится имя нетерминала. Позиция, в которую следует вернуться, указывается на ребрах GSS. В результате в процессе разбора создается и обрабатывается меньше дескрипторов. Пример исходного и соответствующего ему модифицированного стека представлен на рис. 1.



GSS в исходном GLL



GSS в библиотеке Iguana

Рис. 1: Пример из статьи [1]: представление GSS для грамматики $A ::= aAb \mid aAc \mid d$ и входной строки aac

Реализация, созданная авторами статьи, называется Iguana и написана на Java. На рис. 2 показана IDEF0 диаграмма ключевого компонента данной библиотеки.

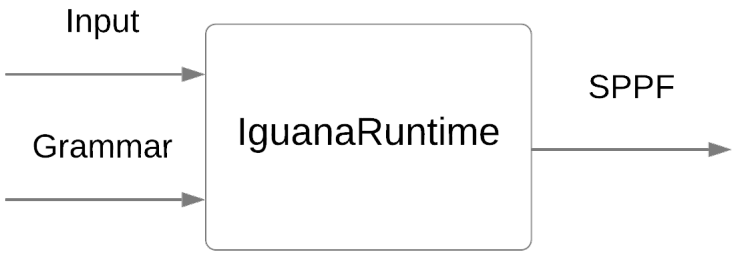


Рис. 2: Высокоуровневая архитектура библиотеки Iguana в нотации IDEF0

Эта библиотека является оптимизированной версией GLL-анализатора. Поэтому в данной работе предлагается использование этого эффективного инструмента, решающего задачу синтаксического анализа линейного входа, для поддержки запросов с контекстно-свободными ограничениями для графов.

1.7. Графовая база данных Neo4j

Neo4j является наиболее используемой графовой СУБД [9]. Данные в Neo4j представляются в виде узлов (вершин) и отношений между ними (ребер). Также и у вершин, и у ребер могут храниться свойства.

Каждое свойство состоит из ключа и значения. Помимо этого, вершинам и ребрам можно сопоставлять метки.

Для того чтобы осуществлять запросы к базе данных из Java, можно использовать Cypher Java API или Native Java API (рис. 3). Исполь-

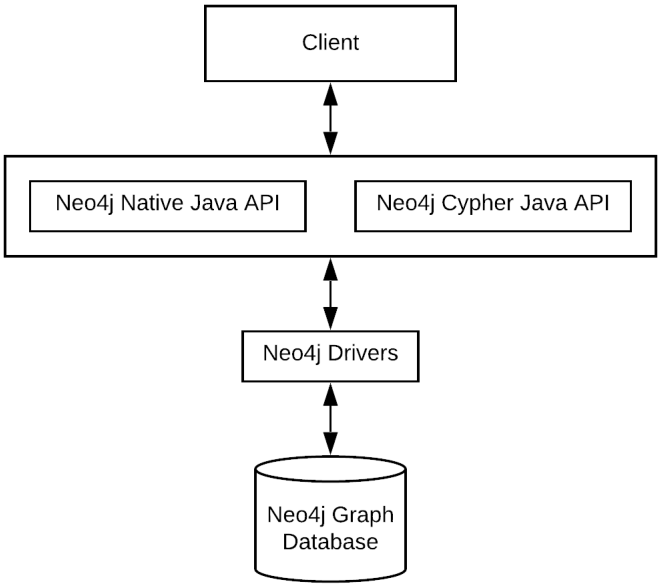


Рис. 3: Архитектура Neo4j Java API

зование представленных API позволяет из кода создавать встроенную базу данных, а не подключаться к ней как к внешнему сервису.

Так же как и Iguana, Neo4j реализована на Java и является проектом с открытым исходным кодом. В связи с этими преимуществами в данной работе выбрана именно эта графовая база данных.

2. Постановка задачи

Целью данной работы является реализация алгоритма поиска путей с ограничениями, заданными контекстно-свободной грамматикой, для графовой базы данных Neo4j. Для достижения поставленной цели необходимо решить задачи, описанные ниже.

- Модифицировать исходный код библиотеки Iguana для поддержки входных данных представленных в виде графа.
- Интегрировать модифицированную версию Iguana с графовой базой данных Neo4j.

  Взаимодействие с Neo4j должно осуществлятся через ее программный API, то есть не подразумевается интеграция с Cypher.
- Провести экспериментальное исследование на тестовых данных[2].

  Целью исследования является проверка гипотезы о том, что при использовании оптимизированного обобщенного LL-анализатора для поиска путей с контекстно-свободными ограничениями происходит улучшение производительности в сравнении с существующими подходами (например, Meerkat [12] — реализация поиска путей с контекстно-свободными ограничениями на основе парсер-комбинаторов).

[2]Набор тестовых данных: `https://github.com/JetBrains-Research/CFPQ_Data`. Дата обращения: 29.05.2020

3. Модификация библиотеки Iguana

В данном разделе представлена итоговая архитектура модифицированной библиотеки Iguana. Далее описаны основные изменения в логике алгоритма синтаксического разбора, лежащего в основе данной библиотеки, которые были произведены для поддержки графового входа.

На рис. 4 показана диаграмма основных классов Iguana после того, как эта библиотека была модифицирована. Голубым цветом на этой диаграмме отмечены классы, которые на данном этапе работы были изменены или добавлены.

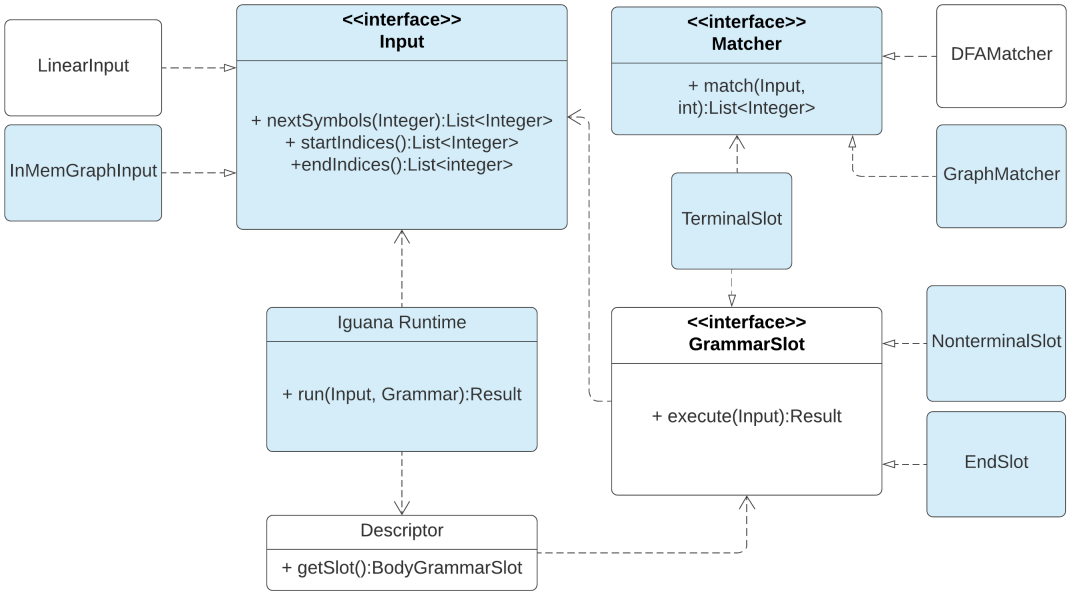


Рис. 4: Диаграмма классов Iguana после модификаций для поддержки графового входа

Процесс синтаксического анализа в Iguana устроен следующим образом: в методе `run` класса `IguanaRuntime` инициализируется очередь дескрипторов. Далее из очереди последовательно берутся дескрипторы. Обработка слота, полученного из текущего дескриптора, зависит от вида данного слота.

Для того чтобы поддержать обработку входных данных, представленных в виде графа, потребовалось изменить абстракцию входа. Поскольку в графе стартовых вершин может быть много, изменения также

были внесены в инициализацию очереди дескрипторов. Для поддержки контекстно-свободных запросов для графа поменялось и поведение при обработке различных видов слотов. Помимо этого, модификации коснулись внутренних структур, использующихся в GLL, это GSS для хранения стека и SPPF для компактного представления полученных деревьев вывода.

Далее будет подробно показано, каким образом описанные изменения были сделаны.

3.1. Входные данные

Входные данные в исходном коде проекта Iguana представлялись интерфейсом `Input`. Его реализация хранила в себе массив символов, представляющих собой линейный вход — строку. Одним из ключевых методов в данном классе являлся метод, принимающий позицию во входной строке и возвращающий символ на следующей позиции.

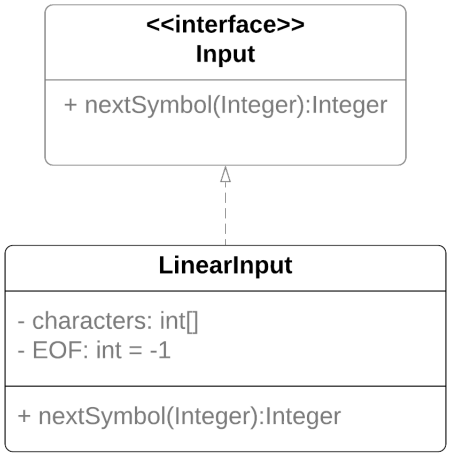


Рис. 5: Диаграмма классов, представляющих входные данные до модификаций

В процессе работы была добавлена новая реализация `InMemGraphInput` интерфейса `Input`, представляющая собой граф в виде списка смежности, а также набора стартовых и финальных вершин искомых путей. При использовании этого класса граф хранится в оперативной памяти. В случае графа текущую позицию во входных данных характеризует

идентификатор вершины. Поэтому для реализации `InMemGraphInput` описанный выше метод был модифицирован следующим образом: теперь он принимает идентификатор вершины и возвращает список меток на ребрах, выходящих из вершины с данным идентификатором. В связи с этим в ходе изменений сигнатура метода в интерфейсе была изменена: возвращаемое значение стало списком символов. Соответственно, в реализации для строк возвращаемым значением теперь является одноэлементный список.

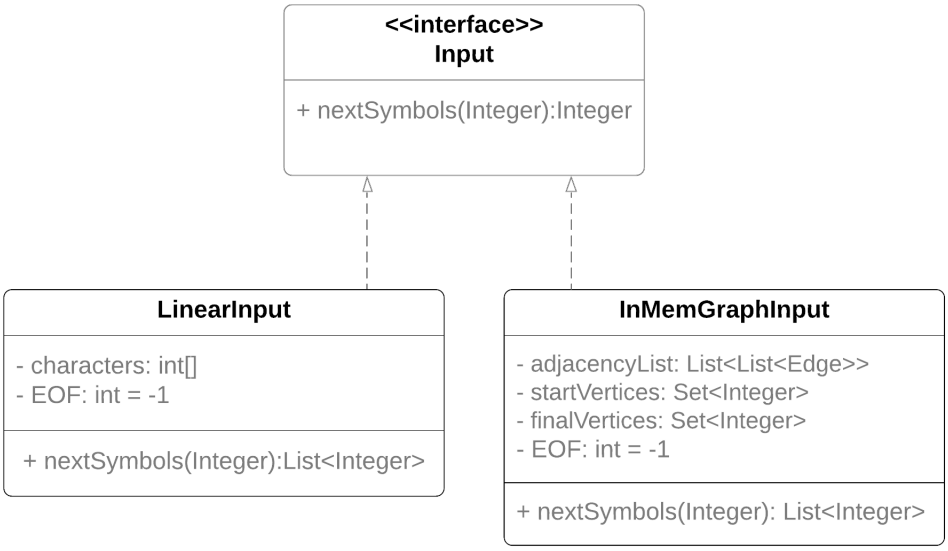
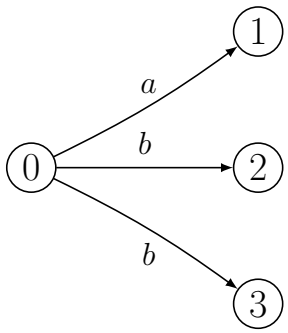


Рис. 6: Диаграмма классов, представляющих входные данные после модификаций

3.2. Интерфейс сопоставления входа с терминалом

При обработке дескриптора с терминальным слотом (то есть слотом вида $N \to \alpha \ . \ x\beta$, где x — терминал грамматики), в Iguana используется метод функционального интерфейса `Matcher`. Данный метод делает следующее: он принимает индекс во входной строке i (содержащийся в текущем обрабатываемом дескрипторе) и возвращает такой индекс j, что подстрока входной строки от i до $j - 1$ соответствует терминалу x. То есть это тот индекс во входной строке, откуда надо продолжать синтаксический разбор, перейдя к слоту $N \to \alpha x \ . \ \beta$.

В графе подобных позиций может быть несколько. Рассмотрим следующий пример:



Для терминала b и текущего идентификатора во входной строке равного 0, можно перейти как к вершине с идентификатором 2, так и 3. Таким образом, сигнатура данного метода в интерфейсе `Matcher` изменилась, поскольку теперь он возвращает список идентификаторов.

3.3. Поведение в слотах

Обработка дескриптора зависит от того, какой тип слота содержится в данном дескрипторе.

- В случае терминального слота в Iguana с помощью сопоставления входа с терминалом x вычислялась новая позиция j во входной строке. Поскольку для строк эта позиция была одна, то дальше сразу же вызывалась обработка нового дескриптора (со слотом вида $N \to \alpha x\ .\ \beta$ и новой позицией j во входе).

  Для графа сопоставление входа с терминалом возвращает список позиций (вершин в графе), на которые можно перейти по данному терминалу. Поэтому теперь при обработке терминального слота создаются новые дескрипторы для каждой позиции. Далее они добавляются в очередь дескрипторов и обработка текущего дескриптора заканчивается.

- В случае нетерминального слота (то есть слота вида $N \to \alpha\ .\ x\beta$, где x — нетерминал грамматики) поведение практически не поменялось, поскольку оно было построено на использовании метода

`nextSymbols` интерфейса `Input` и все необходимые изменения были заключены в нем.

- В случае слота вида $N \to x$. (то есть нетерминал N был обработан) в библиотеке Iguana осуществлялась проверка на то, что следующий символ во входе принадлежит множеству $FOLLOW(N)$. Для корректной обработки конца входной строки к ней был добавлен символ EOF. Этим символом было дополнено множество $FOLLOW$ для каждого нетерминала.

  Для того чтобы поддержать данную логику при работе с графом, идейно каждой вершине из множества финальных вершин было добавлено мнимое исходящее ребро с меткой EOF. Технически это означает, что возвращаемое значение метода `nextsymbols` класса `InMemGraphInput` было дополнено символом EOF для финальных вершин.

3.4. Сжатое представление леса разбора

Сжатое представление леса разбора (SPPF) содержит в себе все деревья вывода. При работе с графами эта структура позволяет восстановить все найденные пути. Вершины в графе SPPF бывают четырех типов.

- Терминальные вершины — вершины вида (i, T, j), где T — терминал грамматики, а i и j обозначают начальную и конечную позицию во входе.

- Нетерминальные вершины — по аналогии с терминальными вершинами имеют вид (i, N, j) (N в данном случае нетерминал грамматики).

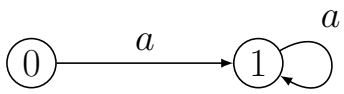
- Упакованные вершины имеют форму $(N \to \alpha\ .\ \beta, j)$. Они используются для поддержания нескольких альтернативных деревьев вывода.

- Промежуточные вершины описываются слотом, начальной и конечной позицией, то есть они имеют вид $(i, N \to \alpha \ . \ \beta, j)$. Данный тип вершин необходим для бинаризации SPPF.

В Iguana метод получения сжатого представления леса разбора генерировал SPPF без учета альтернатив, то есть он строил лишь одно возможное дерево разбора. При этом все альтернативы мемоизировались в процессе работы. Для того чтобы получить SPPF с альтернативами, в данной библиотеке использовался шаблон проектирования Visitor. При этом результат разбора для вершины зависит от результатов для ее потомков. Таким образом, при наличии цикла в SPPF возникает проблема: результат для вершины в цикле зависит от результатов самой себя. В Iguana случай цикла в сжатом представлении леса разбора не обрабатывался.

В случае графового входа цикл в SPPF может быть. Это означает, что путей, удовлетворяющих запросу с ограничениями в виде заданной контекстно-свободной грамматики, бесконечное количество.

Рассмотрим граф:



Сжатое представление леса разбора для него показано на рис. 7.

Для того чтобы поддерживать циклический SPPF, в ходе работы был добавлен класс-обертка, хранящийся внутри класса `VisitResult` (в нем содержится результат обхода). В начале обхода какой-либо вершины SPPF v в качестве ее результата сохраняется обертка над фиктивным результатом. Затем обрабатываются потомки v. Допустим, в процессе обработки обнаруживается цикл, то есть для новой рассматриваемой вершины u потомком является исходная v. Тогда u находит мемоизированное для v значение результата и сохраняет себе ссылку на него. Когда все потомки исходной вершины v будут обработаны, их результаты агрегируются, и фиктивный результат заменяется на посчитанный. Поскольку изменяется только значение, а не ссылка, это

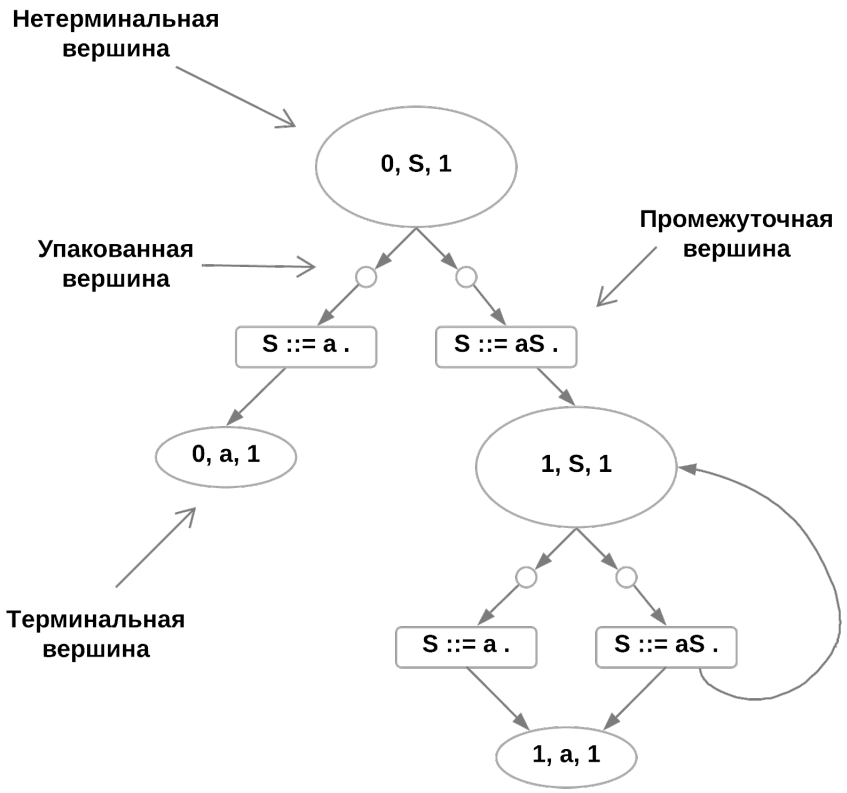


Рис. 7: Сжатое представление леса разбора для данного графа и грамматики $S \to aS \mid a$

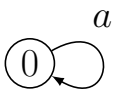
изменение отражается на всех вершинах, хранящих ссылку на результат исходной вершины v.

3.5. Представление стека

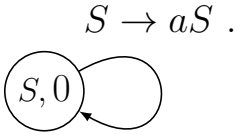
Для хранения вершин стека в Iguana используется интерфейс `GSSNode`. При инициализации начальных дескрипторов, соответствующих нулевой позиции в строке и правилам вида $S \to . \, x$, где S — стартовый нетерминал грамматики, в рассматриваемой библиотеке создается вершина стека специального вида — стартовая GSS вершина (класс `StartGSSNode`). Эта сущность отличается от обычной вершины стека (класс `DefaultGSSNode`) тем, что она мемоизирует результат (текущий SPPF), и тем, что из нее не могут исходить ребра в другие GSS вершины.

При входных данных, представленных в виде графа, исходящие ребра из стартовой GSS вершины являются допустимыми. Рассмотрим

пример. Пусть задан граф:



И дана грамматика с единственным нетерминалом S, терминалом a и правилами $S \to a$ и $S \to aS$. В таком случае GSS принимает следующий вид:



Отсутствие исходящего ребра из единственной вершины этого стека привело бы к некорректной работе алгоритма. Этот пример иллюстрирует, почему от класса `StartGSSNode` необходимо было отказаться. Теперь для начальных GSS вершин также используется реализация `DefaultGSSNode`, к которой была добавлена мемоизация результата.

4. Интеграция модифицированной Iguana с графовой базой данных Neo4j

Полученное решение было интегрировано с Neo4j. Общение с базой осуществляется при помощи Native Java API. Это позволило реализовать всю необходимую функциональность, используя простой интерфейс взаимодействия с базой. Методы, которые понадобились для интеграции с Neo4j, представлены на рис. 8. База данных при этом является встраиваемой.

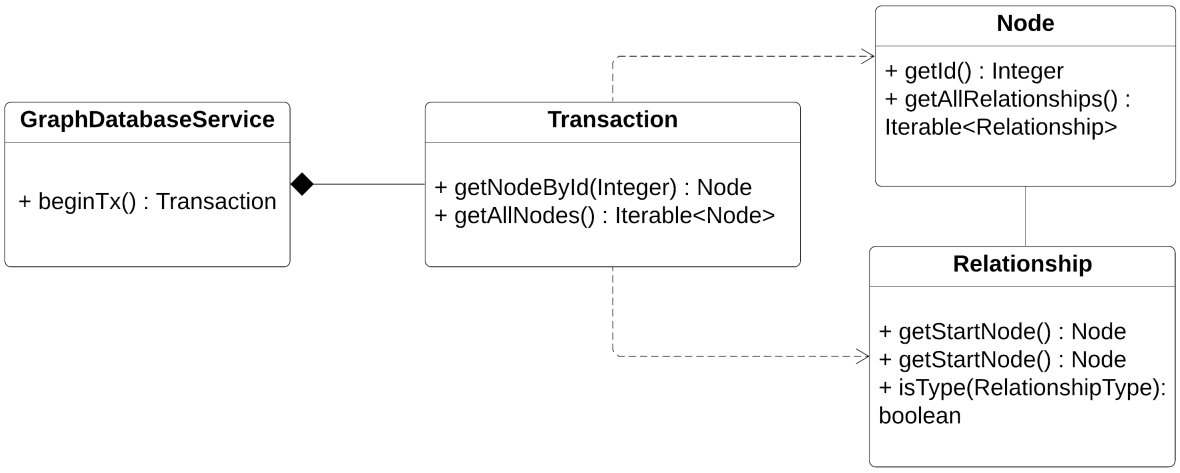


Рис. 8: Диаграмма классов интерфейса, предоставляемого Neo4j Native Java API

В результате интеграции был добавлен абстрактный класс `GraphInput`. Его наследует прежняя реализация графового входа, в которой граф хранится в оперативной памяти. Для интеграции с Neo4j добавилась еще одна реализация данного абстрактного класса — `Neo4jGraphInput`. Полем этого класса является `GraphDatabaseService`, с помощью которого осуществляются транзакции к базе. Помимо этого, при создании `Neo4jGraphInput` пользователь передает функцию получения терминала по ребру.

Итоговая диаграмма классов, отвечающих за входные данные, выглядит следующим образом:

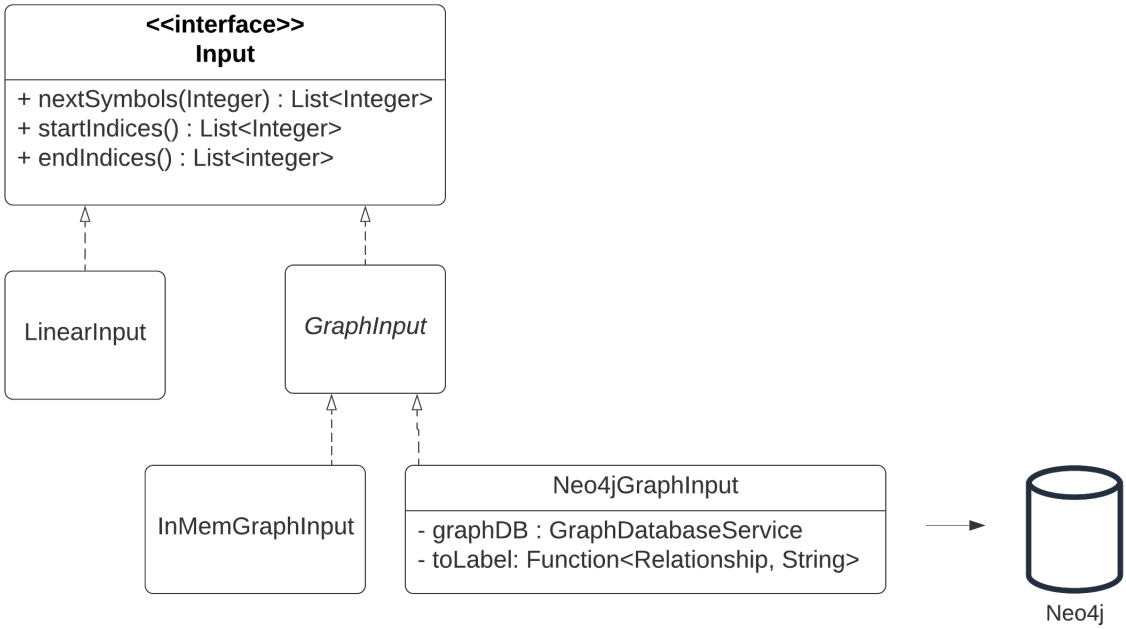


Рис. 9: Диаграмма классов, которые представляют входные данные, после интеграции с Neo4j

5. Эксперимент

В данном разделе описывается экспериментальное исследование полученной реализации.

5.1. Данные для исследования

В качестве данных для эксперимента использовались графы, описанные ниже.

- Enzyme — это граф с данными о белковых последовательностях. В данном графе 15 тыс. вершин и 48 тыс. ребер.

- Geospecies — это граф, содержащий в себе таксономическую иерархию и географическую информацию о видах животных. Именно на этом графе проводились эксперименты в статье [5]. По результатам исследования авторы заключили, что пока что КС запросы не применимы на практике из-за времени выполнения этих запросов. Граф Geospecies содержит 225 тыс. вершин и 1.6 млн ребер.

Датасеты с этими графами доступны в формате представления данных RDF. Для того чтобы добавить их в базу данных, использовался плагин к Neo4j под названием neosemantics[3].

В данных графах есть ребра с метками нескольких типов, одни из них — `skos__broaderTransitive` (bt), `skos__narrowerTransitive` (nt), `rdfs__subClassOf` (sc) и `rdf__type` (t).

На основе этих типов было сделано три запроса. Это были вариации запроса поиска потомков одного поколения для ребер с различными метками. Для этого были заданы следующие грамматики.

- G_1: грамматика с нетерминалом S, терминалами nt и nt^{-1} и правилом вывода $S \to nt\ S\ nt^{-1} \mid nt\ nt^{-1}$. Метка nt^{-1} используется для обозначения перехода по ребру с меткой nt в обратном направлении.

[3]Плагин neosemantics: `https://github.com/jbarrasa/neosemantics`. Дата обращения: 29.05.2020

- G_2: аналогичная грамматика с терминалами bt, bt^{-1}, а также правилом вывода $S \to bt\ S\ bt^{-1} \mid bt\ bt^{-1}$.

- G_3: грамматика с нетерминалом S, терминалами sc, sc^{-1}, t, t^{-1} и правилом вывода $S \to sc\ S\ sc^{-1} \mid t\ S\ t^{-1} \mid sc\ sc^{-1} \mid t\ t^{-1}$.

5.2. Сравнение с библиотекой Meerkat

Эксперимент проводился на сервере со следующими характеристиками:

- операционная система Ubuntu 18.04;

- процессор Intel Core i7-6700 CPU, 3.40GHz;

- объем оперативной памяти 64 Гб.

Для сравнения с библиотекой Meerkat в качестве множества стартовых вершин бралась каждая вершина по отдельности, а финальными считались все вершины графа.

Результаты замеров производительности и потребления памяти для графа Enzyme для каждого из запросов приведены на рис. 11. Каждой стартовой вершине соответствует одна точка на графике — по оси абсцисс отложено количество найденных уникальных путей из данной стартовой вершины. Это количество было посчитано с помощью обхода SPPF.

Гистограмма распределения длин всех найденных путей на графе Enzyme представлена на рис. 10. Максимальная длина пути оказалась равна восьми.

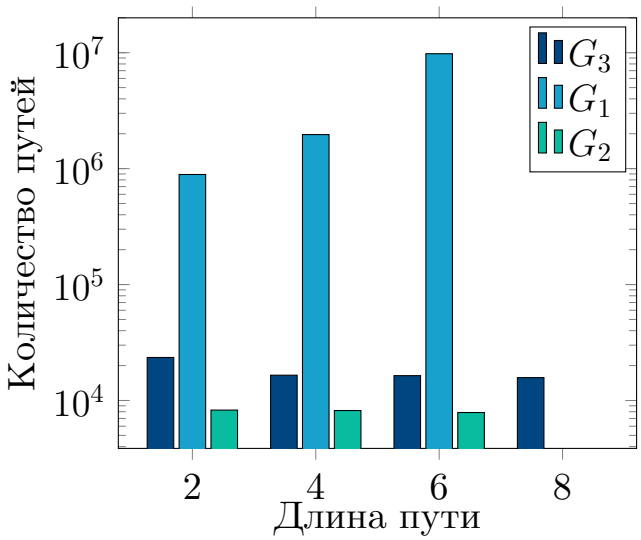
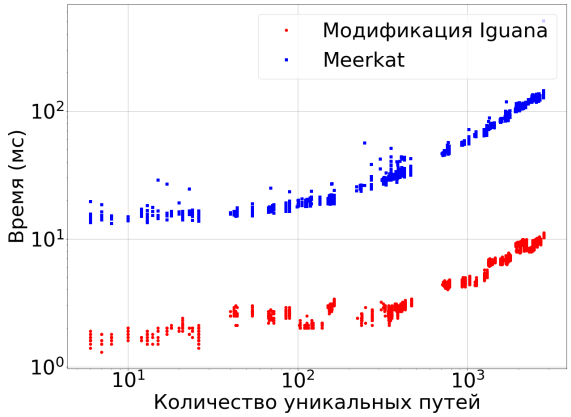


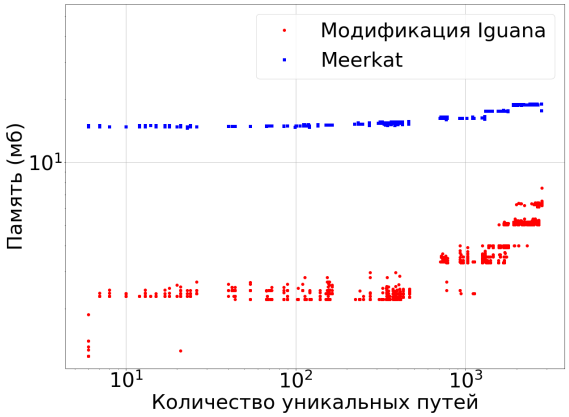
Рис. 10: Распределение длин найденных путей (Enzyme)

На запросе G_1 модифицированная Iguana показала на порядок лучшие результаты, чем Meerkat — и время выполнения запроса, и память были уменьшены почти в 10 раз. На запросах G_2 и G_3 с некоторого момента при увеличении количества путей происходит "взрыв" — резко начинают расти показатели времени и памяти. Однако параллельно с этим медленный рост также сохраняется. Это является достаточно странным поведением и нуждается в дальнейшем анализе. Тем не менее, на основной массе вершин полученная реализация превосходит по производительности Meerkat.

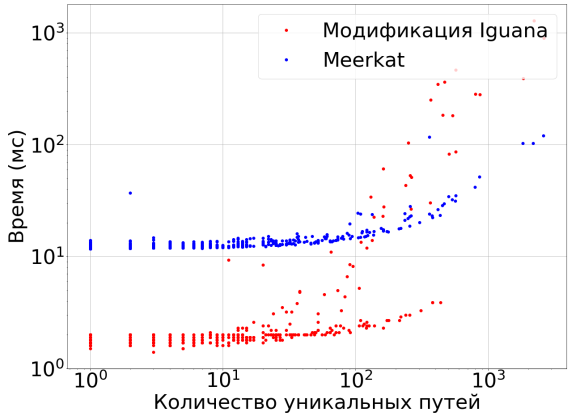
Дальнейшие эксперименты были произведены на графе Geospecies. Он значительно больше Enzyme в размерах. Гистограмма распределения длин найденных путей в данном графе представлена на рис. 12. Их количество в разы превышает количество путей в графе Enzyme.



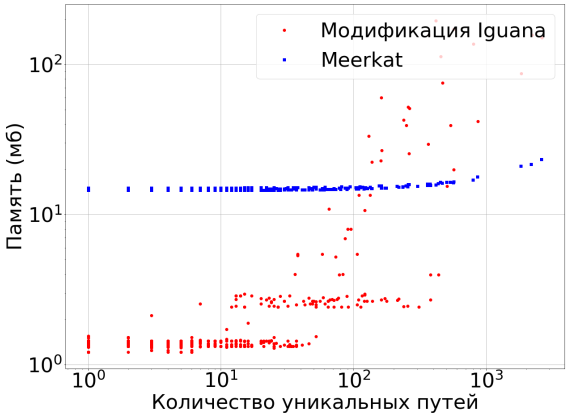
(a) время для запроса G_1



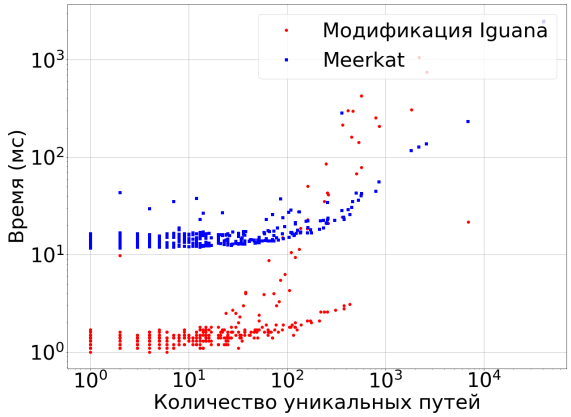
(b) память для запроса G_1



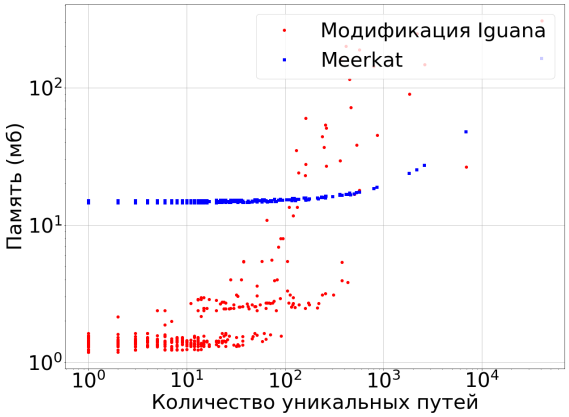
(c) время для запроса G_2



(d) память для запроса G_2



(e) время для запроса G_3



(f) память для запроса G_3

Рис. 11: Результаты замеров на графе Enzyme

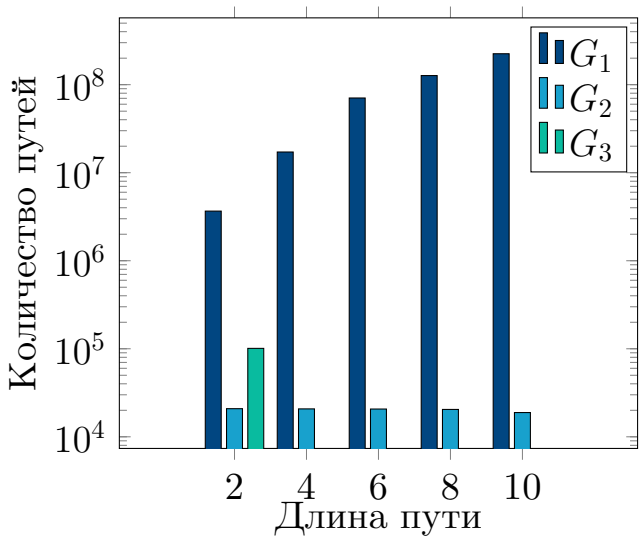
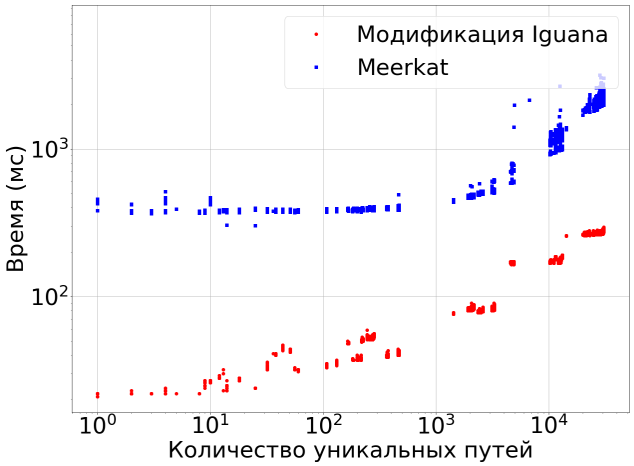


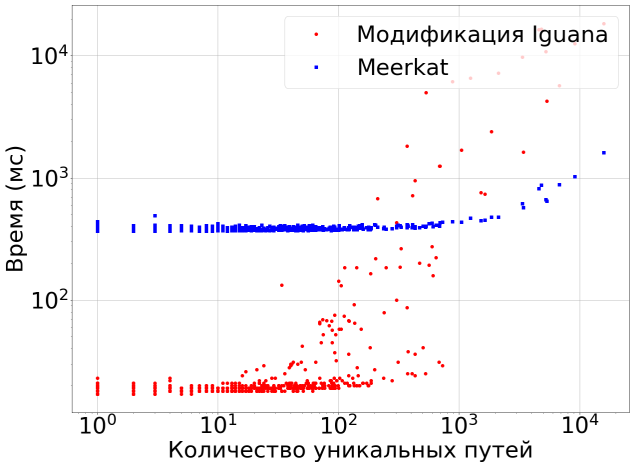
Рис. 12: Распределение длин найденных путей (Geospecies)

Результаты экспериментов на графе Geospecies представлены на рис. 13. На всех запросах Iguana продемонстрировала улучшение производительности по сравнению с Meerkat. Даже при малом количестве путей запрос для одной вершины в Meerkat работает на этом графе не менее 300 мс. В Iguana такого не наблюдается: несмотря на размер графа, запросы, в которых ответом служит небольшое количество путей, исполняются почти на два порядка быстрее, чем при использовании Meerkat. Однако на запросе G_2 для Geospecies присутствуют аналогичные выбросы, как и на запросах G_2, G_3 для графа Enzyme.

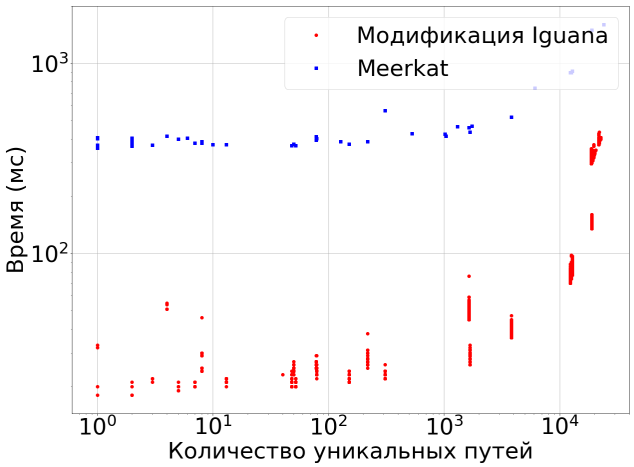
Таким образом, проведенные эксперименты показывают, что на реальных данных полученная реализация в большинстве случаев демонстрирует существенный прирост производительности. При этом на некоторых запросах данная реализация нуждается в дальнейшем исследовании.



(a) время выполнения для запроса G_1



(b) время выполнения для запроса G_2



(c) время выполнения для запроса G_3

Рис. 13: Результаты замеров на графе Geospecies

Заключение

При выполнении данной работы получены следующие результаты.

- Библиотека Iguana расширена для поддержки входных данных, представленных как в виде графа, так и строк.
- Расширенная версия Iguana была интегрирована с графовой базой данных Neo4j. Встроенная база создается из кода с помощью Neo4j Native Java API.
- Проведено экспериментальное исследование и сравнение с Meerkat на RDF данных Enzyme и Geospecies. Было показано, что реализованный подход является жизнеспособным и перспективным. На одних запросах решение дало улучшение производительности на порядок, на других были выявлены проблемы с выбросами, которые можно дальше анализировать.

Дальнейшими возможными направлениями для развития работы являются:

- Проведение экспериментального исследования на еще более реальных данных, то есть таких, где не только граф, но и сама грамматика возникают при решении прикладных задач. Примером таких данных является граф MemoryAliases [19].
- Полноценная интеграция полученного решения с Neo4j, в том числе с языком запросов Cypher.

Список литературы

[1] Afroozeh Ali, Izmaylova Anastasia. Faster, Practical GLL Parsing // International Conference on Compiler Construction / Springer. — 2015. — 04. — P. 89–108.

[2] Azimov Rustam, Grigorev Semyon. Context-Free Path Querying by Matrix Multiplication // Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences Systems (GRADES) and Network Data Analytics (NDA). — GRADES-NDA '18. — New York, NY, USA : Association for Computing Machinery, 2018. — 10 p.

[3] Azimov R., Grigorev S. Path Querying with Conjunctive Grammars by Matrix Multiplication // Programming and Computer Software. — 2019. — 12. — Vol. 45, no. 7. — P. 357–364.

[4] Chaudhary Anoop, Faisal Abdul. Role of graph databases in social networks. — 2016. — 06.

[5] An Experimental Study of Context-Free Path Query Evaluation Methods / Jochem Kuijpers, George Fletcher, Nikolay Yakovets, Tobias Lindaaker. — 2019. — 07. — P. 121–132.

[6] Grigorev Semyon, Avdyukhin Dmitry. Relaxed parsing of regular approximations of string-embedded languages // Perspectives of System Informatics - 10th International Andrei Ershov Informatics Conference, PSI 2015, Revised Selected Papers / Ed. by Manuel Mazzara, Andrei Voronkov. — Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). — Germany : Springer, 2016. — 1. — P. 291–302.

[7] Grigorev Semyon, Ragozina Anastasiya. Context-Free Path Querying with Structural Representation of Result // Proceedings of the 13th Central Eastern European Software Engineering Conference in

Russia. — CEE-SECR '17. — New York, NY, USA : Association for Computing Machinery, 2017. — 7 p.

[8] Have Christian Theil, Jensen Lars Juhl. Are graph databases ready for bioinformatics? // Bioinformatics (Online). — 2013. — 10.

[9] Kemper Chris. Beginning Neo4j. — 1st edition. — USA : Apress, 2015.

[10] Medeiros Ciro M., Musicante Martin A., Costa Umberto S. Efficient Evaluation of Context-Free Path Queries for Graph Databases // Proceedings of the 33rd Annual ACM Symposium on Applied Computing. — SAC '18. — New York, NY, USA : Association for Computing Machinery, 2018. — P. 1230–1237.

[11] Neo4j's Graph Query Language: An Introduction to Cyphe. — URL: `https://neo4j.com/developer/cypher-basics-i/` (online; accessed: 29.05.2020).

[12] Parser Combinators for Context-Free Path Querying / Ekaterina Verbitskaia, Ilya Kirillov, Ilya Nozkin, Semyon Grigorev // Proceedings of the 9th ACM SIGPLAN International Symposium on Scala. — Scala 2018. — New York, NY, USA : Association for Computing Machinery, 2018. — P. 13–23.

[13] Reps Thomas. Program Analysis via Graph Reachability // Proceedings of the 1997 International Symposium on Logic Programming. — ILPS '97. — Cambridge, MA, USA : MIT Press, 1997. — P. 5–19.

[14] Robinson Ian, Webber Jim, Eifrem Emil. Graph Databases: New Opportunities for Connected Data. — 2nd edition. — O'Reilly Media, Inc., 2015. — ISBN: 1491930896.

[15] Santos Fred C., Costa Umberto S., Musicante Martin A. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases // Web Engineering. — Cham : Springer International Publishing, 2018. — P. 225–233.

[16] Scott Elizabeth, Johnstone Adrian. GLL Parsing // Electron. Notes Theor. Comput. Sci. — 2010. — . — Vol. 253, no. 7. — P. 177–189.

[17] Shemetova Ekaterina, Grigorev Semyon. Path querying on acyclic graphs using Boolean grammars // Proceedings of the Institute for System Programming of RAS. — 2019. — 10. — Vol. 31. — P. 211–226.

[18] Tomita Masaru. An Efficient Context-Free Parsing Algorithm for Natural Languages // Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2. — IJCAI'85. — San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1985. — P. 756–764.

[19] Zheng Xin, Rugina Radu. Demand-Driven Alias Analysis for C.