Санкт-Петербургский государственный университет

Программная инженерия

Погожельская Влада Владимировна

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

Отчет по производственной практике

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

Санкт-Петербург
2021

Оглавление

Введение

Графовая модель данных широко используется во многих областях [12], таких как, например, графовые базы данных [2], биоинформатика [11], статический анализ кода [5] и т.д. Основное преимущество данной модели над реляционной моделью данных заключается в том, что получение информации об отношении между объектами в ней выполняется очень быстро, поскольку взаимосвязи между узлами не вычисляются во время выполнения запроса, а хранятся в самой модели. Одной из самых распространенных задач, связанной с анализом данных, представляемых с помощью графов, является поиск путей. В графовых базах данных для их анализа используются запросы, естественным способом задания которых является наложение ограничений на пути между вершинами.

Одним из подходов для выражения таких запросов является определение формальных грамматик над алфавитом меток ребер. Путь принадлежит языку, задаваемому формальной грамматикой, если ему принадлежит слово, получаемое конкатенацией меток ребер данного пути [14]. Наиболее часто при таком подходе в качестве грамматик берутся регулярные. Так, например, в одной из наиболее распространенных графовых баз данных Neo4j в качестве языка запросов используется декларативный язык Cypher [10], поддерживающий ограничения на пути, заданные в терминах регулярных языков. Причем стоит отметить, что регулярные ограничения поддерживаются лишь частично и на текущий момент язык запросов довольно ограничен. Одним из способов расширить выразительность запросов является переход к ограничениям в терминах контекстно-свободных языков. Они строго расширяют выразительность запросов по сравнению с регулярными языками и тем самым позволяют решать более широкий класс задач. Одной из таких задач является запрос поиска всех потомков одного поколения (same-generation query). Она выразима в терминах контекстно-свободных языков, но не в терминах регулярных выражений [13].

Несмотря на то, что проблема поиска путей, удовлетворяющих огра-

ничениям в терминах формальной грамматики, довольно хорошо изучена и существуют множество различных алгоритмов для ее решения [1, 3, 7, 9], все еще существует ряд проблем, связанных с их применимостью в анализе реальных данных [4]. Одной из наиболее остро стоящих проблем является низкая производительность существующих алгоритмов, поэтому актуальной задачей является разработка и реализация новых алгоритмов, решающих данную проблему.

Одной из недавних разработок является адаптация классического алгоритма синтаксического анализа Generlized LL для выполнения контекстно-свободных запросов на графах[1]. Важно отметить, что полученный алгоритм поддерживает весь класс контекстно-свободных языков. Тем не менее при проведении экспериментального исследования полученного алгоритма при выполнении ряда запросов было выявлено резкое снижение производительности. Модифицированный GLL, так же как и оригинальный алгоритм, возвращает информацию не только о достижимостях между вершинами, но и информацию для построения самих путей. Для этого используется специальная структура данных — сжатое представление леса разбора (SPPF). Однако, данная структура потребляет значительное количество ресурсов и, как следствие, замедляет работу всего алгоритма. На практике же, ограничения на процессорные ресурсы являются весьма существенными, при этом восстановление самих путей не всегда требуется — достаточно получить лишь информацию о самом их существовании. Таким образом, была выдвинута гипотеза о том, что добавление возможности отключения построения SPPF даст ощутимый прирост в скорости выполнения запросов на реальных данных. Данная работа направлена на улучшение производительности разработанного алгоритма и, в частности, на проверку этой гипотезы.

[1]GitHub репозиторий реализации обобщенного GLL алгоритма: `https://github.com/YaccConstructor/iguana`, дата обращения: 15.12.2020

1. Постановка задачи

Целью данной работы является улучшение производительности алгоритма поиска путей с контекстно-свободными ограничениями для графовой базы данных Neo4j.

Для достижения поставленной цели были выделены перечисленные ниже задачи.

- Проведение анализа кода и его рефакторинг с целью выявления и устранения проблем производительности текущей реализации GLL алгоритма.
- Добавление возможности отключения построения SPPF и возвращения информации лишь о достижимостях в графе.
- Проведение экспериментального исследования на реальных данных и сравнение полученного решения с уже существующим.

2. Обзор

В данный раздел включены основные определения с дальнейшим описанием как теоретической части алгоритма, так и его реализации.

2.1. Понятия из теории формальных языков

Введем базовые определения из теории формальных языков, которые будут использоваться в дальнейшем.

Определение 2.1. Контекстно-свободная грамматика — это четверка $\langle N, \Sigma, P, S \rangle$, где

- N — набор нетерминальных символов;
- Σ — набор терминальных символов, $\Sigma \cap N = \varnothing$;
- P — набор правил или продукций вида $N_i \to \alpha$, где $N_i \in N$ и $\alpha \in \{\Sigma \cup N\}^* \cup \varepsilon$;
- $S \in N$ — стартовый нетерминал.

Определение 2.2. Язык, задаваемый контекстно-свободной грамматикой $\mathbb{G}$ — множество строк, выводимых в грамматике

$$L(\mathbb{G}) = \{w \in \Sigma^* | S \Rightarrow^* w\},$$

где $A \Rightarrow^* w$ обозначает, что строка $w \in \Sigma^*$ может быть получена из нетерминала A с помощью некоторой последовательности продукций из P.

Определение 2.3. Язык, задаваемый контекстно-свободной грамматикой с указанным стартовым нетерминалом A — множество строк, выводимых в грамматике из нетерминала A

$$L(\mathbb{G}_A) = \{w \in \Sigma^* | A \Rightarrow^* w\}.$$

2.2. Задача поиска путей в графе с контекстно-свободными ограничениями

Пусть даны:

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

Рассмотрим путь в графе $\mathbb{D}$:

$$\pi = (e_0, e_1, \cdots, e_{n-1}, e_n),$$

где $e_k = (v_{k-1}, t_k, v_k),\ v_i \in V,\ t_k \in T$. Путям в графе сопоставим слово $l(\pi) = t_1 t_2 \cdots t_n$ — конкатенацию меток на ребрах данного пути. Тогда, если рассматривать искомую задачу, то окажется, что результирующее множество путей в графе задает множество слов, иначе говоря — язык.

Во введенных обозначениях могут быть сформулированы следующие задачи.

- **Задача поиска путей в графе с контекстно-свободными ограничениями** заключается в том, чтобы найти все такие пути в графе, что $l(\pi) \in L(\mathbb{G})$ и $v_0 \in V_S,\ v_n \in V_F$.
- **Задача достижимости в графе с контекстно-свободными ограничениями** заключается в поиске множества пар вершин, для которых найдется путь с началом и концом в этих вершинах, что слово, составленное из меток рёбер пути, будет принадлежать заданному языку: $\{(v_i, v_j) \mid \exists\ l(\pi) \in L(\mathbb{G})$ и $v_0 \in V_S,\ v_n \in V_F\}$.

Заметим, что при работе с графовой моделью данных довольно часто возникает необходимость выявления сложных зависимостей в ней и,

в зависимости от контекста и области применения, оба варианта представленных выше задач имеют важное практическое значение.

2.3. Обобщенный GLL алгоритм

Одним из классических алгоритмов синтаксического разбора является LL(k) алгоритм. Он выполняет нисходящий анализ с предпросмотром. Иначе говоря, решение о том, какую продукцию грамматики следует применить, основывается на просмотре k следующих за текущим символом. По сравнению с алгоритмами восходяшего анализа данный алгоритм проще в процессе написания и отладки, так как полностью соответствует структуре грамматики. Однако, он применим только к подмножеству класса контекстно-свободных грамматик и не поддерживает неоднозначные контекстно-свободные грамматики, а также грамматики, использующие левую рекурсию.

Существует еще один класс синтаксических анализаторов — обобщенные анализаторы, которые применяются для обработки неоднозначных грамматик. К таким анализаторам относится Generalized LL (GLL) — алгоритм обобщенного нисходящего анализа. В отличие от LL(k) алгоритма, где может возникнуть ситуация, когда нельзя однозначно определить, какую продукцию необходимо применить в текущем состоянии разбора входной строки, в GLL алгоритме поддерживается очередь дескрипторов. Каждый дескриптор представляет собой структуру, описывающую текущее состояние анализатора. Таким образом, с помощью очереди дескрипторов в процессе работы парсера рассматриваются все возможные варианты переходов.

В таблице синтаксического анализа для алгоритма обобщенного анализа может храниться несколько альтернатив для разбора текущего нетерминала. В таком случае может происходить дублирование дескриптора. Для эффективного хранения и переиспользования множества различных дескрипторов в GLL используется специальная структура — Graph Structured Stack (GSS) [15].

Также важно отметить, что данный алгоритм решает задачу по-

иска не просто достижимостей, а путей. Для этого в GLL как часть текущего состояния разбора входных данных поддерживается сжатое представление леса разбора, содержащее в себе все деревья вывода.

Описанный GLL алгоритм был обобщён с обработки линейного входа на обработку графов, как это было описано в работе [6]. Для этого в статье были предложены следующие модификации.

- Запрос теперь представляет собой тройку: множество начальных вершин, множество конечных вершин, грамматика.
- Исходное множество дескрипторов должно включать в себя все стартовые вершины графа.
- На шаге перехода к следующему символу необходимо поддерживать все возможные варианты перехода, которые соответствуют всем исходящим рёбрам вершины.
- В случае завершения разбора необходимо осуществлять проверку принадлежности конечной вершины разбора множеству конечных вершин графа.

Описанные принципы работы обобщенного GLL алгоритма важны для понимания особенностей его реализации, о которой речь пойдет ниже.

2.4. Обзор существующего решения

Реализация решения задачи поиска путей в графе с контекстно-свободными ограничениями базируется на библиотеке Iguana[2], написанной на Java. Библиотека предоставляет реализацию классического GLL алгоритма, на основе которой и было получено обобщение на графы. Преимущество Iguana состоит с вот, что в ней используется модификация стека GSS, которая позволяет свести вычислительную и пространственную сложность GLL алгоритма к кубическим величинам от размера входа.

[2]GitHub репозиторий библиотеки Iguana: `https://github.com/iguana-parser/iguana`, дата обращения: 15.12.2020

В контексте данной работы важно обратить внимание на следующие внесенные изменения в процесс работы алгоритма GLL для обобщения на графы.

- Для того чтобы поддержать обработку входных данных, представленных в виде графа, потребовалось изменить абстракцию входных данных. Была добавлена новая реализация интерфейса входных данных Input, представляющая собой список смежности графа и набор стартовых и финальных вершин искомых путей.

- Так как для входных данных в виде графа, в отличие от линейного входа, стартовых вершин может быть много, были внесены изменения в инициализацию очереди дескрипторов. При обработке дескриптора со слотом $(N \to \alpha \ . \ x\beta)$, где x — терминал, в Iguana использовался метод nextSymbols, принимающий индекс во входной строке i и возвращающий индекс j такой, что подстрока входной строки от i до $j{-}1$ соответствует x. Таким образом, j — индекс во входной строке, откуда надо продолжать синтаксический разбор, перейдя к слоту $(N \to \alpha x \ . \ \beta)$. Так как в графе подобных позиций может быть несколько, то и сигнатура данного метода была изменена таким образом, что теперь он возвращает список идентификаторов.

В качестве хранилища для графов была использована графовая база данных Neo4j — наиболее часто используемая графовая СУБД [8]. Neo4j поддерживает язык запросов Cypher, а данные в ней представляются в виде узлов (вершин) и отношений между ними (ребер). Вершинам и ребрам можно сопоставлять метки. Так же как и Iguana, Neo4j реализована на Java и является проектом с открытым исходным кодом. Модифицированный алгоритм был интегрирован с Neo4j при помощи Native Java API.

3. Экспериментальное исследование существующего решения

В дипломной работе Анны Власовой были проведены эксперименты на реальных графах и сделан сравнительный анализ с библиотекой Meerkat[3]. Данная библиотека поддерживает запросы с контекстно-свободными ограничениями на основе парсер-комбинаторов. Она также использует графовую базу Neo4j в качестве хранилища графов. Проведенный сравнительный анализ показал, что на реальных данных алгоритм в большинстве случаев дает существенный прирост в производительности, а значит является правильным направлением в решении задачи поиска путей в графе с контекстно-свободными ограничениями.

Однако, в ряде случаев при запросе поиска в графе путей от одной вершины до всех было выявлено неожиданное ухудшение поведения полученного решения. Так как причина проблем с производительностью осталась неясна, в рамках данной работы было принято решение проведения повторных экспериментов на более широком множестве запросов.

Экспериментальное исследование, описанное в данном разделе, а также все дальнейшие эксперименты в работе проводились на одном или двух графах, а именно:

- Enzyme — граф о белковых последовательностях (48 тыс. вершин и 86 тыс. ребер);
- Geospecies — граф о таксономической иерархии видов животных (450 тыс. вершин и 2.2 млн ребер).

Они были взяты из набора данных $CFPQ\_Data$[4], собранного исследователями лаборатории языковых инструментов Jetbrains Research.

В данных графах есть ребра с метками нескольких типов, одни из них — broaderTransitive (bt), subClassOf (sc) и type (t). На основе них

[3]Репозиторий библиотеки Meerkat.Graph: `https://github.com/YaccConstructor/Meerkat`, дата обращения: 15.12.2020

[4]GitHub репозиторий набора данных $CFPQ\_Data$: `https://github.com/JetBrains-Research/CFPQ_Data`

было сделано три запроса. Эти запросы — вариации запроса поиска потомков одного поколения для ребер с различными метками. Для них были сформированы следующие контекстно-свободные грамматики G_1, G_2 и G_3 соответственно.

$$\begin{aligned} S \rightarrow & \overline{subClassOf}\ \ S\ subClasOf \mid \overline{type}\ \ S\ type \\ & \mid \overline{subClassOf}\ \ subClassOf \mid \overline{type}\ type \end{aligned} \tag{1}$$

$$S \rightarrow \overline{subClassOf}\ \ S\ subClasOf \mid subClassOf \tag{2}$$

$$\begin{aligned} S \rightarrow & broaderTransitive\ \ S\ \overline{broaderTransitive} \\ & \mid broaderTransitive\ \ \overline{broaderTransitive} \end{aligned} \tag{3}$$

В качестве начальных вершин были взяты множества вершин V_i следующего вида:

$$\forall r \in R = \{1,\ 2,\ 4,\ 8,\ 16,\ 32,\ 50,\ 100,\ 500,\ 1000,\ 5000,\ 10000\}$$

$$V(r) = V_1 \sqcup V_2 \sqcup \cdots \sqcup V_m,\ \forall 1 \leq i < m : |V_i| = r,\ |V_m| \leq r,$$

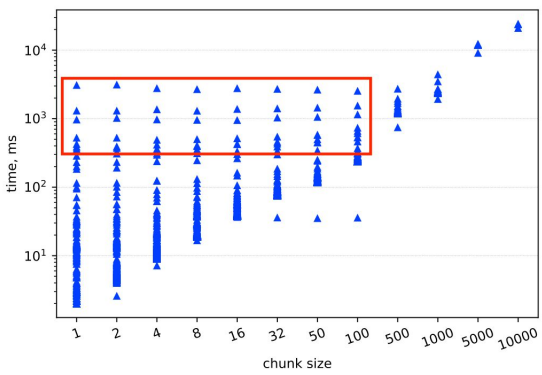
иначе говоря, различные разбиения множества вершин графа на подмножества одинакового размера. В качестве финальных вершин во всех экспериментах рассматривалось всё множество вершин графа.

Все эксперименты проводились на машине со следующими характеристиками и окружением:

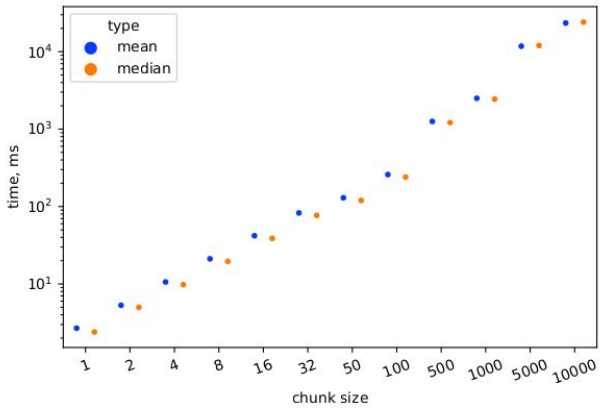
- операционная система Ubuntu 20.04;
- процессор Intel Core i7-6700 CPU, 3.40GHz;
- объем оперативной памяти 32 Гб;
- версия Java 11;
- версия Neo4j 4.0.3.

Результаты замеров производительности для графа Enzyme приведены на рис.1-3. На графиках слева каждому стартовому множеству

вершин соответствует отдельная точка на графике, по оси X отложены их размеры, по оси Y — время работы алгоритма в миллисекундах. На графиках справа представлены медиана (median) и среднее значение (mean) для каждого размера множества стартовых вершин. Разброс медианы и среднего арифметического свидетельствует о наличии отклонений в производительности работы алгоритма.

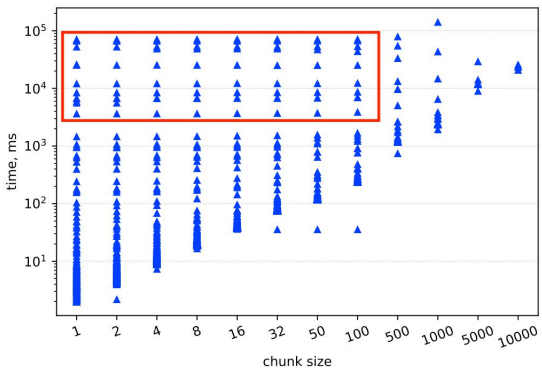


(a) время выполнения запросов

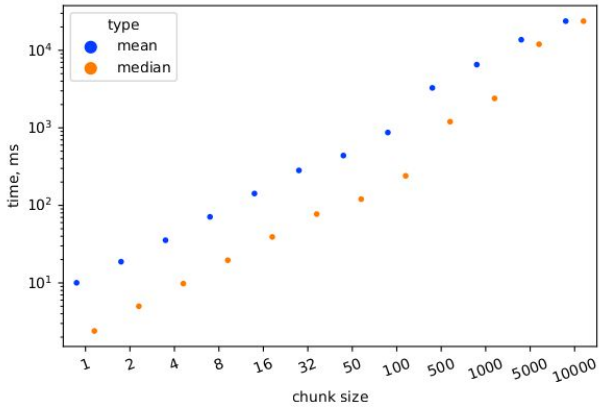


(b) медиана и среднее время выполнения запросов

Рис. 1: Грамматика G_1 на Enzyme

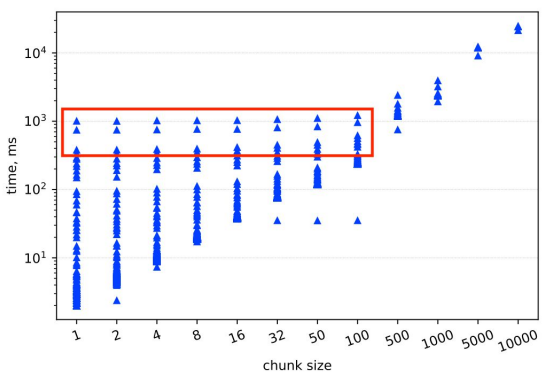


(a) время выполнения запросов

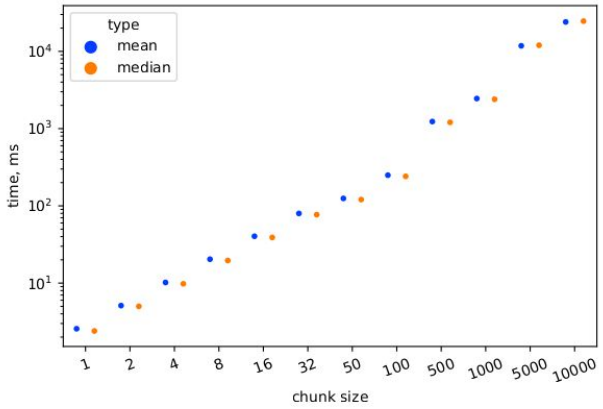


(b) медиана и среднее время выполнения запросов

Рис. 2: Грамматика G_2 на Enzyme



(a) время выполнения запросов



(b) медиана и среднее время выполнения запросов

Рис. 3: Грамматика G_3 на Enzyme

При анализе полученных данных были выделены следующие проблемы.

- Запросы от некоторых множеств вершин выполняются аномально долго по сравнению со скоростью выполнения других запросов того же типа. Для наглядности данные запросы выделены красным прямоугольником. Те же выводы подтверждаются разницей между значениями среднего арифметического и медианы.

- Enzyme является относительно небольшим графом, однако данные эксперименты явно показали, что уже на нем выполнение запросов от крупного множества вершин происходит критически медленно.

Несмотря на показанную в дипломной работе эффективность алгоритма при запросах от одной вершины до всех, данные проблемы в поведении ставят под вопрос его применимость на практике в том виде, в котором он есть.

4. Оптимизация производительности алгоритма

В данном разделе описываются изменения, внесенные в реализацию алгоритма для повышения его производительности и устранения проблем, выявленных в разделе об экспериментальном исследовании существующего решения.

4.1. Результаты профилирования существующего решения

Прежде чем приступить к оптимизациям кода было необходимо изучить распределение производительности алгоритма и выявить его «узкие» места. Самый удобный способ это сделать — запуск алгоритмов с активным профилированием. В качестве инструмента для этого был выбран Java Flight Recorder (JFR), так как он интегрирован в виртуальную машину Java (JVM) и почти не вызывает накладных расходов на производительность. Анализ результатов запусков от «плохих» и от «хороших» множеств вершин производился на основе полученных flame-графиков. Flame-график — диаграмма для трассировки стека, которая отражает процент времени от общего времени работы программы, затраченного в той или иной части кода.

На основе анализа полученных результатов были сделаны выводы о том, что наибольшее количество процессорного времени тратится в основном методе класса Neo4jGraphInput — nextSymbols. Данный метод используется для сопоставления текущего входа с терминалом грамматики. Он принимает идентификатор вершины и возвращает список меток (символов) на ребрах, выходящих из нее. При этом для получения данных меток неизбежно необходимо обратиться напрямую к базе Neo4j. Native Java API предоставляет удобный способ это сделать: получить итератор над множеством исходящих из вершины ребер с помощью метода getRelationships.iterator(). Однако, в результате анализа графика были сделаны выводы о том, что в текущей реализации прак-

тически всё процессорное время затрачивается на вычисления внутри базы. Полученные результаты профилирования послужили основанием для нижеописанного решения.

4.2. Проблема производительности и предложенное решение

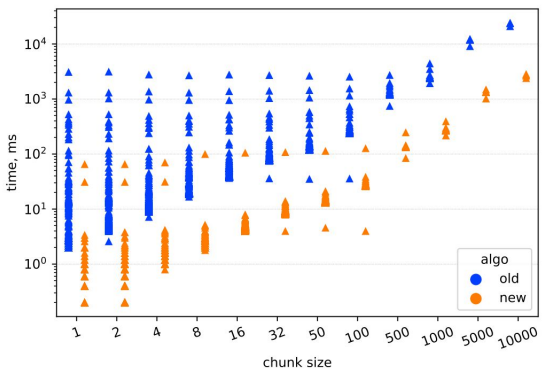
В некоторых случаях алгоритм работал значительно медленнее по следующим причинам. Результатом вызова getRelationships.iterator() является итератор. Иными словами, возвращается не все данные в явном виде, а лишь объект, позволяющий перемещаться по содержимому контейнера. С другой стороны, результат со всеми выходящими метками сохраняется в виде списка в явном виде. Однако, заметим, что после сопоставления входа с терминалом, возможно, далеко не все метки понадобятся для дальнейшей работы алгоритма. Сохранение всех данных приводит к огромным накладным расходам (вплоть до переполнения кучи) в том случае, если степень вершины была очень большая, а большинство меток после соотнесения отбрасывается. Таким образом, необходимо оптимизировать передачу меток из базы данных в дальнейшую обработку.

Довольно распространенным и обоснованным решением данной проблемы является использование Stream API. Stream в Java — поток однотипных данных для однотипной обработки. Иначе говоря, он не является хранилищем данных, а является интерфейсом к источнику, откуда элементы берутся только по мере необходимости. Один из сценариев использования потоков в качестве возвращаемого типа метода выглядит следующим образом. В вызываемом методе надо задать обработку объектов с помощью одной или нескольких промежуточной операций, а в вызывающем — конечную операцию. Метод nextSymbols был переписан в соответствии с данным сценарием для всех типов входных данных. Теперь источником данных является база данных Neo4j, промежуточной операцией — фильтрация ребер по меткам, а конечной — получение меток из возвращаемого методом nextSymbols потока. Та-

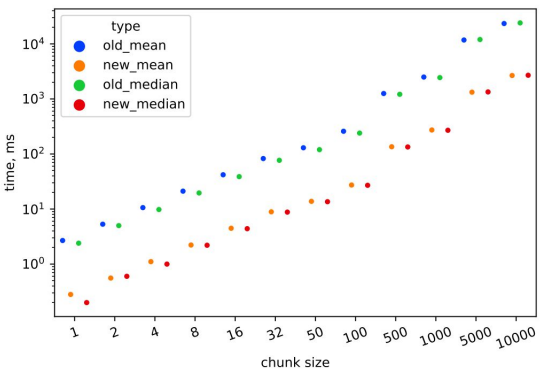
ким образом, в данном методе была обеспечена потоковая обработка данных, извлекаемых посредством обращения к Neo4j.

Преобразованные алгоритмы были протестированы, а затем было запущено повторное профилирование. Его результаты подтвердили то, что внесенные изменения в способ получения меток из базы данных были верными и, тем самым, проблема медленной работы алгоритма при поиске ответа на ряд запросов была устранена. Более того, данные оптимизации в целом повлияли на скорость работы алгоритма в положительную сторону.

Для того чтобы в этом убедиться, рассмотрим результаты замеров предложенной реализации в сравнении с изначальной реализацией алгоритма. Ниже на рис.4-6 отражены сравнительные графики временных замеров. Аналогично, дополнительно выделены медиана и среднее время ответа на запросы для каждого размера множества стартовых вершин. Приведенные результаты показывают, что улучшенная версия алгоритма при работе с реальными данными не только затрачивает на порядок меньше временных ресурсов, но также и в среднем более стабильна. Особенно хорошо это заметно графике медианы и среднего на рис.5, так как для измененного алгоритма эти два показателя практически сравнялись.

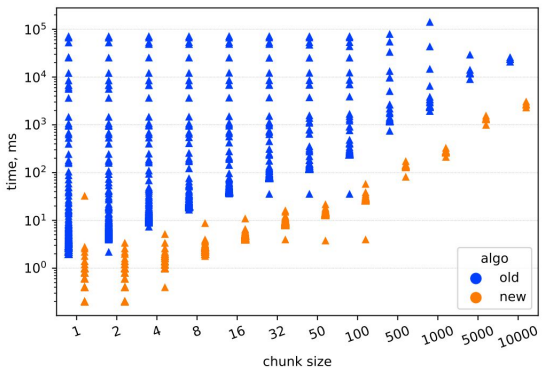


(a) время выполнения запросов

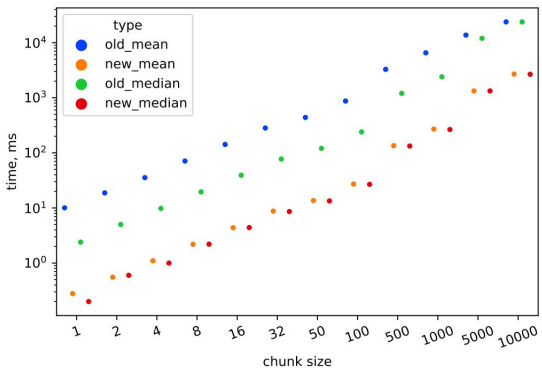


(b) медиана и среднее время выполнения запросов

Рис. 4: Грамматика G_1 на Enzyme

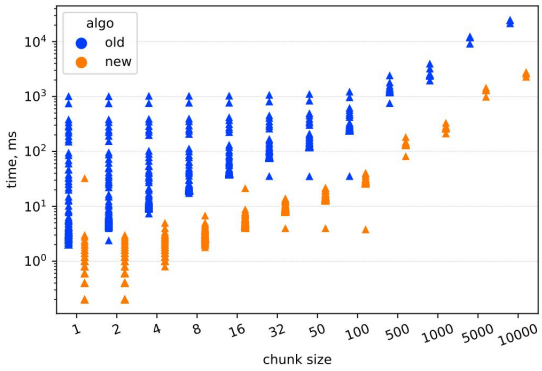


(a) время выполнения запросов

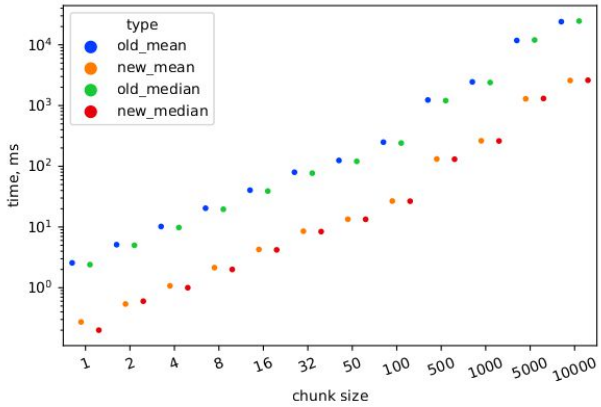


(b) медиана и среднее время выполнения запросов

Рис. 5: Грамматика G_2 на Enzyme



(a) время выполнения запросов



(b) медиана и среднее время выполнения запросов

Рис. 6: Грамматика G_3 на Enzyme

5. Модификация алгоритма GLL

В данном разделе описаны изменения, которые были внесены в работу алгоритма для решения задачи достижимости с ограничениями, заданными в виде контекстно-свободных грамматик.

На рис.7 представлена диаграмма основных классов Iguana после того, как алгоритм был модифицирован и добавлена поддержка получения в качестве результата лишь пар достижимостей. Зеленым цветом на этой диаграмме отмечены классы и методы, которые на данном этапе работы были изменены.

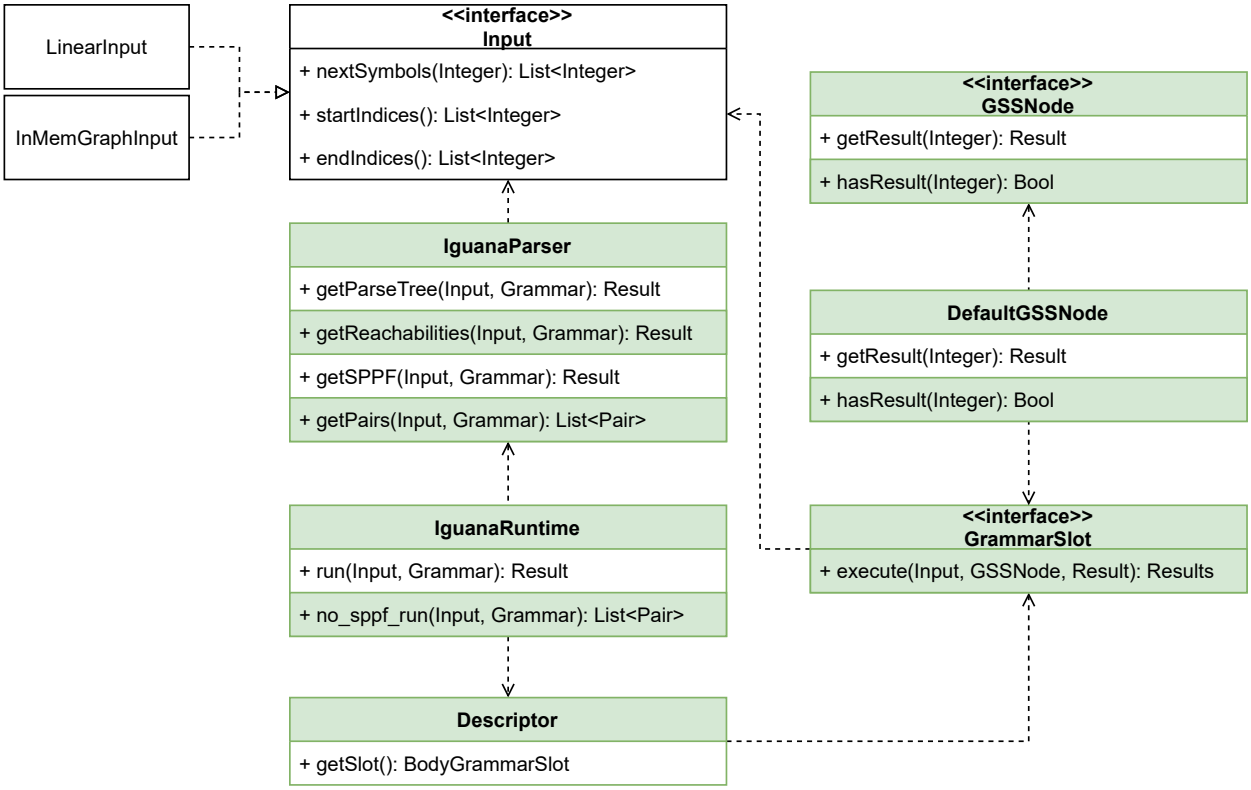


Рис. 7: Диаграмма классов Iguana после модификаций для поддержки опции отключения SPPF

Результатом работы алгоритма GLL на графе является компактное представление леса разбора. Посредством обхода данного дерева можно восстановить пути, удовлетворяющие запросу.

В ходе работы алгоритма при обработке вершины стека GSS создавался текущий узел SPPF, иначе говоря, фрагмент дерева, которое построено на момент создания новой вершины стека. Данная информация

сохранялась на ребрах GSS, после чего полученный дескриптор добавлялся в очередь дескрипторов. В случае решения задачи достижимости необходимость в поддержании SPPF отпадает, так как в качестве результата достаточно возвращать пары начальных и конечных вершин, а сами пути восстанавливать не надо. Поведение алгоритма было модифицировано в описанном выше сценарии соответствующим образом. Создание SPPF вершин и добавление их в дескрипторы предотвращено, а для всех зависимых методов была добавлена новая сигнатура. Таким образом, была расширена функциональность решения и добавлена возможность запуска алгоритма без мемоизации SPPF.

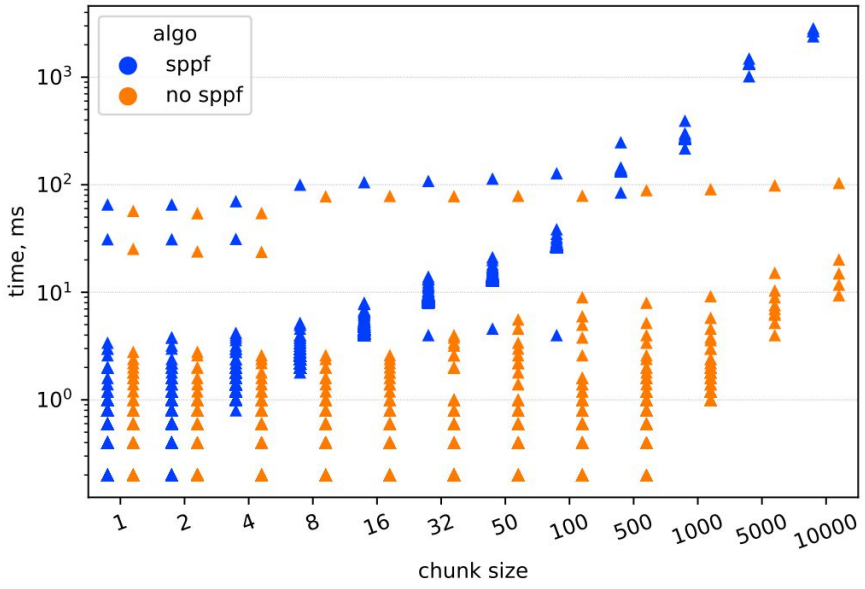
6. Экспериментальное исследование разработанного решения

Полученное решение было протестировано на том же оборудовании. Так как после проведения оптимизаций объем потребляемых процессорных ресурсов значительно уменьшился для обоих алгоритмов, для проведения экспериментального исследования появилась возможность включить в набор данных граф Geospecies.

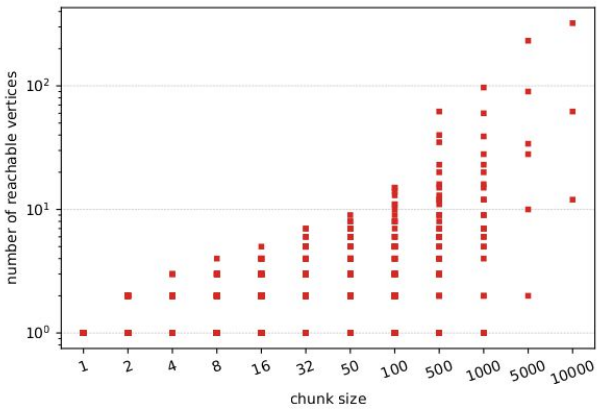
На рис.8-10 приведены результаты замеров производительности для графа Enzyme, а также для каждого запроса соответственно приведены графики зависимостей количества достижимых вершин от стартового множества и графики медиан и среднего времени выполнения запросов. На рис.11-12 приведены аналогичные результаты для Geospecies.

На всех графиках четко прослеживается, что чем крупнее выбранное множество стартовых вершин, тем заметнее становится разница в скорости работы алгоритмов. Время работы алгоритма, который строит SPPF, прямо пропорционально размеру ответа на запрос, так как чем больше путей попадает в ответ, тем больше количество процессорных ресурсов уходит на их мемоизацию. С другой стороны, время работы алгоритма, который не стоит SPPF, практически не увеличивается по мере роста количества возвращаемых пар вершин, что означает, что он может быть применен для решения задачи достижимости на реальных графах.

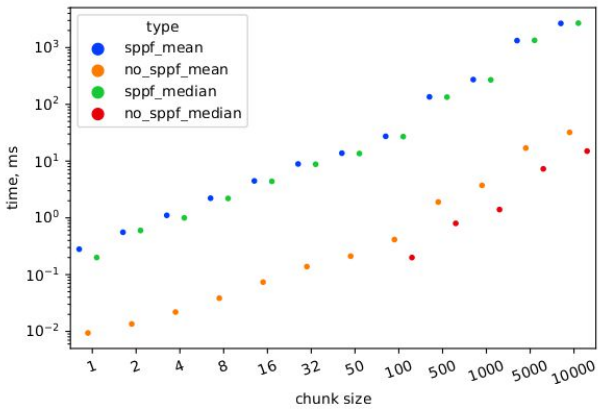
Анализ полученных результатов показал, что модифицированный алгоритм GLL без построения SPPF может быть эффективно применен для решения задачи поиска достижимостей в графе с контекстно-свободными ограничениями на реальных данных. Полученные результаты делают актуальными дальнейшие исследования, направленные как на улучшение данного алгоритма и реализации, так и на полноценную интеграцию его в графовую базу данных Neo4j.



(a) время выполнения запросов

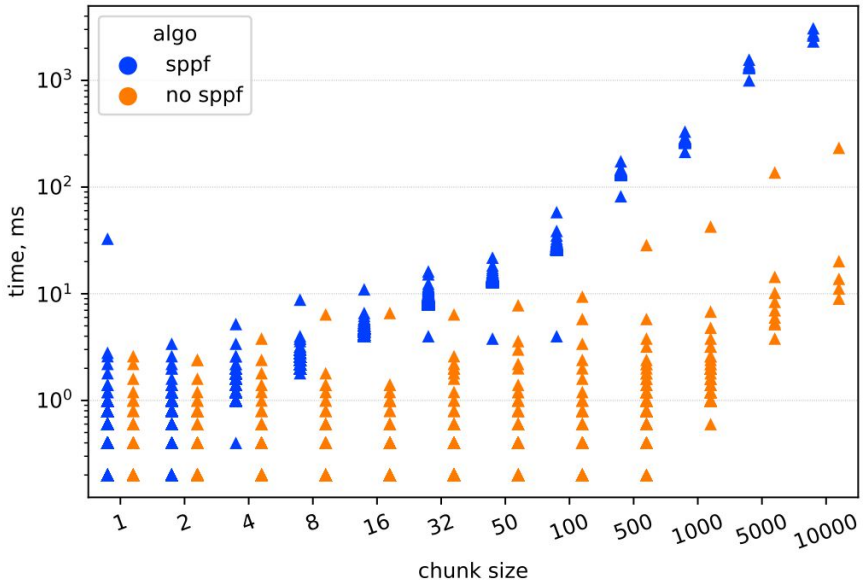


(b) размер ответов на запросы

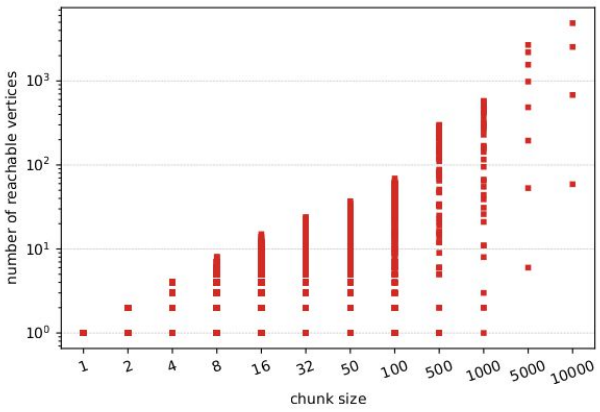


(c) медиана и среднее время на запросы

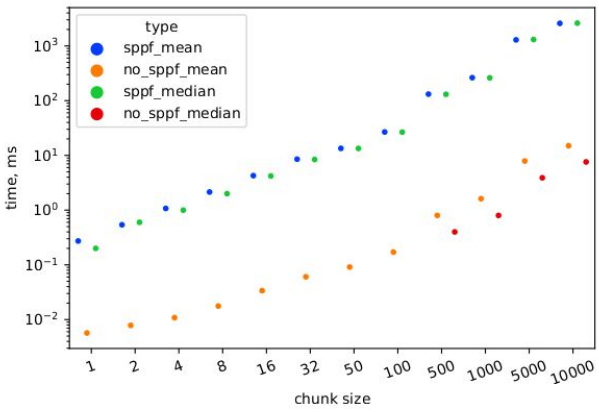
Рис. 8: Грамматика G_1 на Enzyme



(a) время выполнения запросов

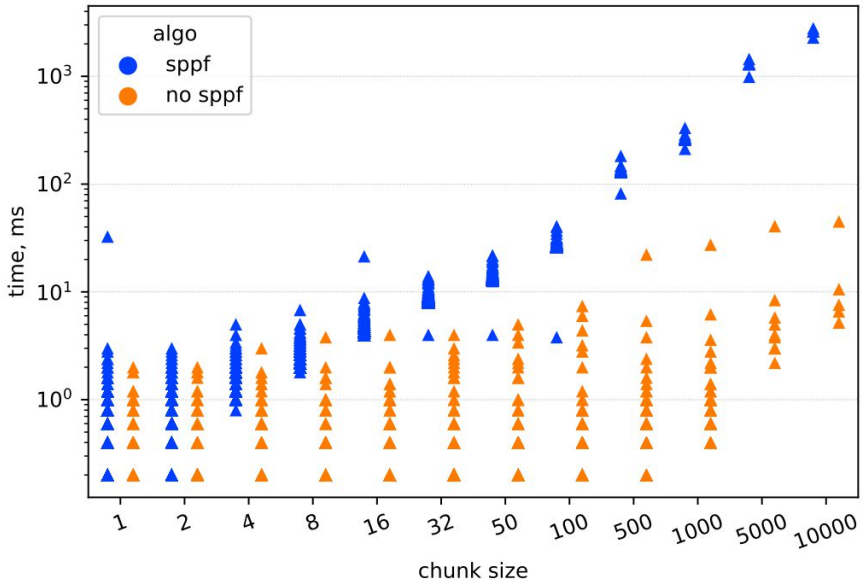


(b) размер ответов на запросы

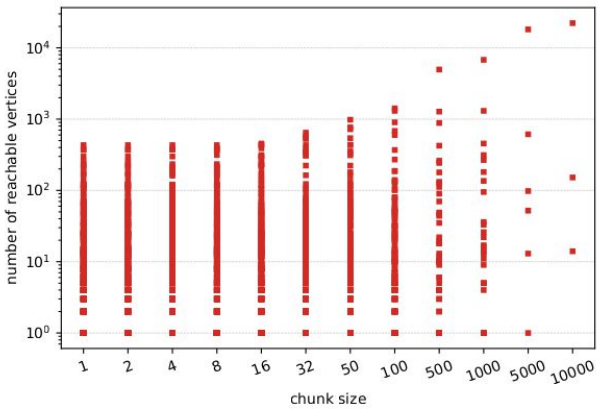


(c) медиана и среднее время на запросы

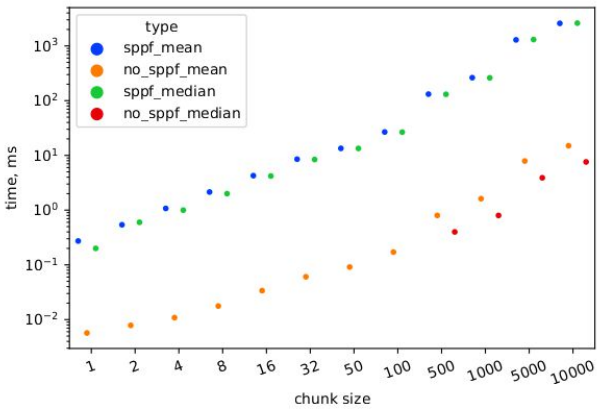
Рис. 9: Грамматика G_2 на Enzyme



(a) время выполнения запросов

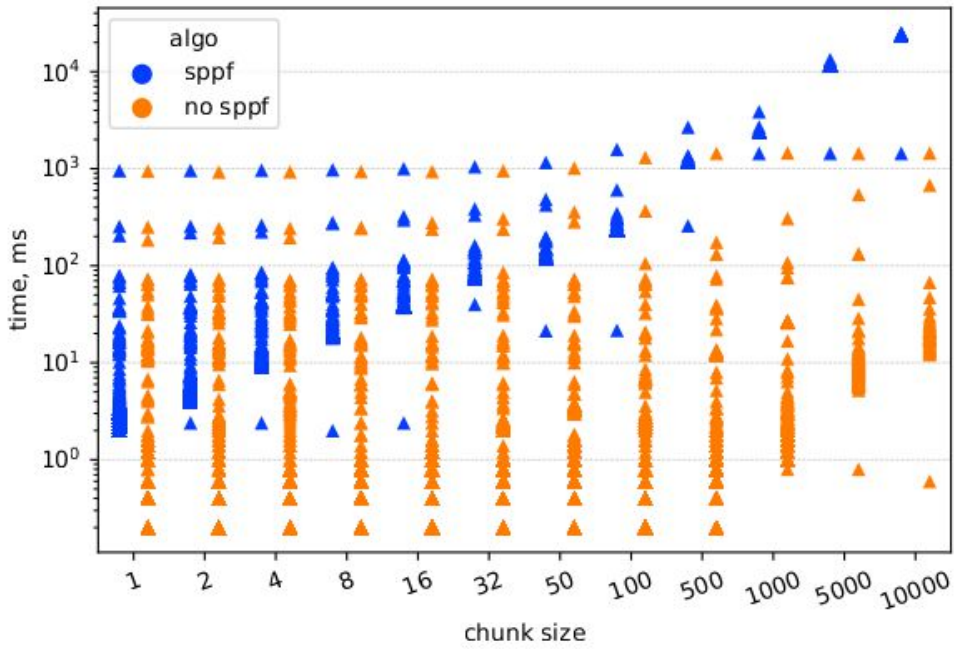


(b) размер ответов на запросы

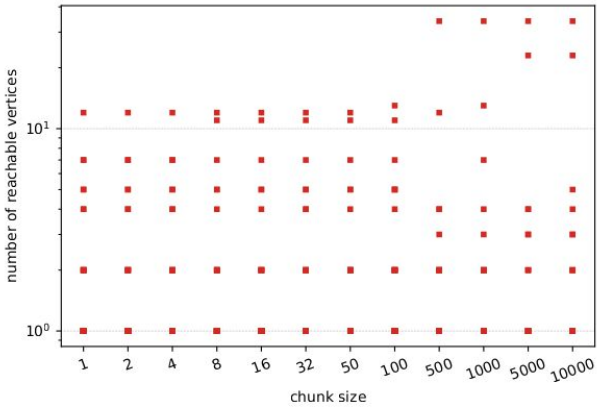


(c) медиана и среднее время на запросы

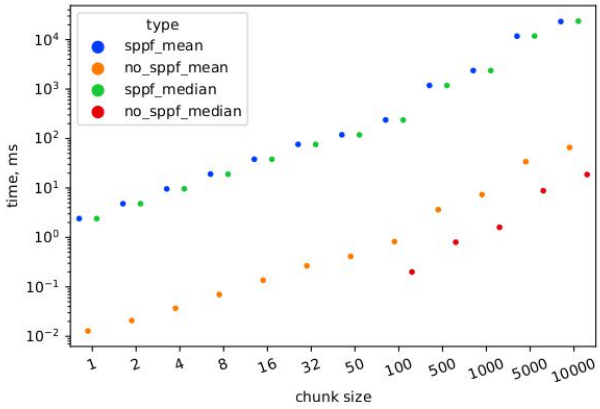
Рис. 10: Грамматика G_3 на Enzyme



(a) время выполнения запросов

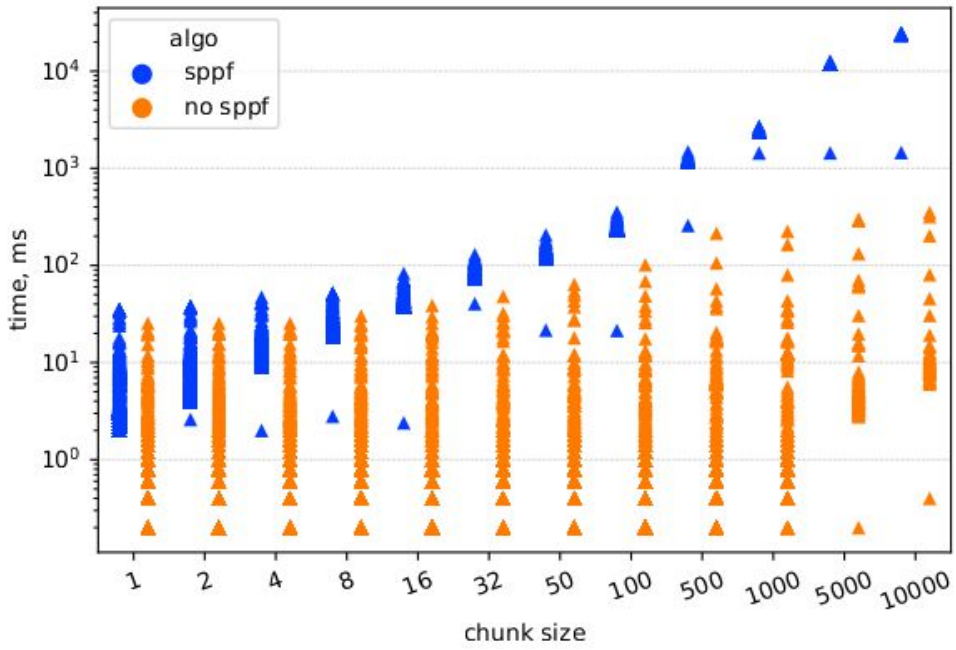


(b) размер ответов на запросы

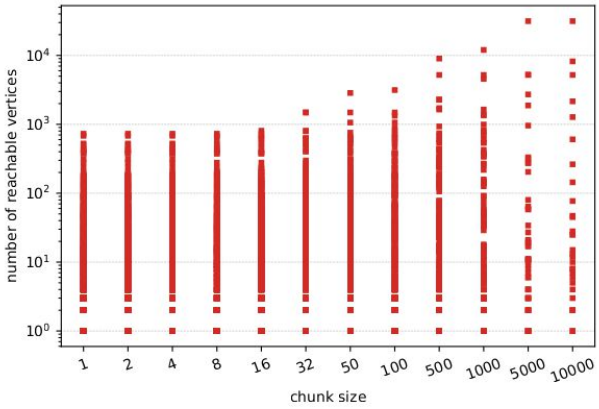


(c) медиана и среднее время на запросы

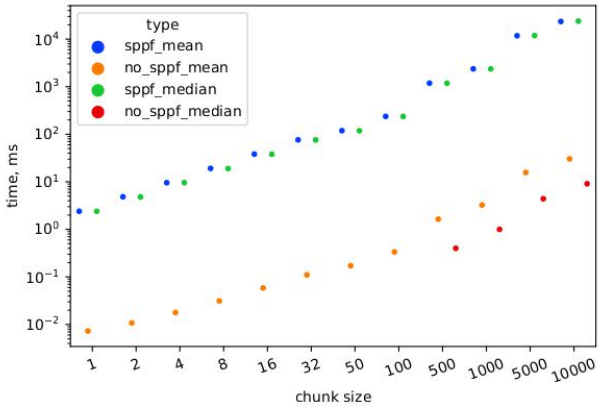
Рис. 11: Грамматика G_1 на Geospecies



(a) время выполнения запросов



(b) размер ответов на запросы



(c) медиана и среднее время на запросы

Рис. 12: Грамматика G_3 на Geospecies

7. Заключение

В рамках производственной практики были выполнены следующие задачи.

- Проведен анализ кода и его рефакторинг.
- Выявлены и устранены проблем производительности текущей реализации GLL алгоритма.
- Добавлена возможность отключения построения SPPF и возврата информации лишь о достижимостях в графе.
- Проведено экспериментальное исследование на реальных данных и сравнение полученного решения с уже существующим.

Дальнейшими направлениями для развития работы являются:

- сравнение полученного решения для Neo4j с другими графовыми базами данных на регулярных запросах;
- сравнение полученного решения для Neo4j с другими графовыми базами данных и самостоятельными инструментами на контекстно-свободных запросах в рамках задачи статического анализа кода;
- публикация алгоритма и результатов экспериментального исследования.

Список литературы

[1] Azimov Rustam, Grigorev Semyon. Context-Free Path Querying by Matrix Multiplication. — GRADES-NDA '18. — New York, NY, USA : Association for Computing Machinery, 2018. — Access mode: https://doi.org/10.1145/3210259.3210264.

[2] Barceló Baeza Pablo. Querying Graph Databases // Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems. — PODS '13. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 175–188. — Access mode: https://doi.org/10.1145/2463664.2465216.

[3] Zhang Xiaowang, Feng Zhiyong, Wang Xin et al. Context-Free Path Queries on RDF Graphs. — 2016. — 1506.00743.

[4] An Experimental Study of Context-Free Path Query Evaluation Methods / Jochem Kuijpers, George Fletcher, Nikolay Yakovets, Tobias Lindaaker // Proceedings of the 31st International Conference on Scientific and Statistical Database Management. — SSDBM '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 121–132. — Access mode: https://doi.org/10.1145/3335783.3335791.

[5] Fast Algorithms for Dyck-CFL-Reachability with Applications to Alias Analysis / Qirun Zhang, Michael R. Lyu, Hao Yuan, Zhendong Su // SIGPLAN Not. — 2013. — Vol. 48, no. 6. — P. 435–446. — Access mode: https://doi.org/10.1145/2499370.2462159.

[6] Grigorev Semyon, Ragozina Anastasiya. Context-Free Path Querying with Structural Representation of Result // Proceedings of the 13th Central amp; Eastern European Software Engineering Conference in Russia. — CEE-SECR '17. — New York, NY, USA : Association for Computing Machinery, 2017. — Access mode: https://doi.org/10.1145/3166094.3166104.

[7] Hellings Jelle. Querying for Paths in Graphs using Context-Free Path Queries. — 2016. — 1502.02242.

[8] Kemper Chris. Beginning Neo4j. — Apress, 2015. — ISBN: 9781484212271.

[9] Medeiros Ciro M., Musicante Martin A., Costa Umberto S. An Algorithm for Context-Free Path Queries over Graph Databases. — 2020. — 2004.03477.

[10] Neo4j's Graph Query Language: An Introduction to Cyphe. — Access mode: `https://neo4j.com/developer/cypher/` (online; accessed: 2020-12-14).

[11] Quantifying variances in comparative RNA secondary structure prediction / James Anderson, Adám Novák, Zsuzsanna Sükösd et al. // BMC bioinformatics. — 2013. — 05. — Vol. 14. — P. 149.

[12] Robinson I., Webber J., Eifrem E. Graph Databases: New Opportunities for Connected Data. — O'Reilly Media, 2015. — ISBN: 9781491930847.

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

[14] 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.

[15] 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.