Санкт-Петербургский государственный университет

АБЗАЛОВ Вадим Игоревич

Выпускная квалификационная работа

Реализация и экспериментальное исследование GLL-парсера, основанного на рекурсивном автомате

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2019 «Программная инженерия»*

Научный руководитель:
доцент кафедры информатики, кандидат физико-математических наук Григорьев С.В.

Рецензент:
академический консультант, кандидат физико-математических наук Азимов Р.Ш.

Санкт-Петербург
2023

Saint Petersburg State University

Vadim Abzalov

Bachelor's Thesis

Implementation and experimental study of the GLL-parser based on a recursive automaton

Education level: bachelor

Speciality *09.03.04 «Software Engineering»*

Programme *CB.5080.2018 «Software Engineering»*

Scientific supervisor:
C.Sc., docent Semyon Grigorev

Reviewer:
academic advisor, C.Sc. of Physics and Mathematics Rustam Azimov

Saint Petersburg
2023

Оглавление

Введение

Представление данных в виде помеченных графов находит широкое применение во многих научных областях [17], таких как анализ социальных сетей [2], биоинформатика[16], статический анализ кода [7] и т.д. Одним из основных преимуществ данной модели над реляционной моделью является более быстрое получение информации об отношениях между объектами, ведь взаимосвязи между узлами не вычисляются во время выполнения запроса, а хранятся в самой модели.

При работе с такими данными зачастую возникают запросы навигации и поиска путей между вершинами. Одним из способов выражения такого рода запросов является определение формальных грамматик над алфавитом меток ребер графа [3]. Путь в графе удолетворяет ограничениям, задаваемым данной формальной грамматикой, если ему принадлежит слово, получаемое конкатенацией меток на ребрах данного пути. Наибольшее практическое применение находят регулярные грамматики, несколько менее популярными являются контекстно-свободные свободные грамматики. Однако, с точки зрения приложений к языкам программирования и компиляции наиболее важными являются именно контекстно-свободные грамматики. Таким образом встает вопрос о необходимости разработки и реализации алгоритмов поиска путей с контекстно-свободными ограничениям (англ. «Context-Free Path Querying», кратко CFPQ).

Несмотря на то, что проблема выполнения контестно-свободных запросов широко изучена и существует целый ряд алгоритмов, ее решающих [1, 4, 10, 13], тем не менее одной из наиболее остро стоящих проблем является низкая производительность существующих алгоритмов [6], поэтому актуальной задачей является разработка и реализация новых алгоритмов, решающих данную проблему.

Одним из недавно разработанных методов решения данной задачи является алгоритм, использующий матричное представление для получения информации о достижимостях в графе. Данный алгоритм был предложен Азимовым Р.Ш. в [5] и основан на матричных операци-

ях. В его работе было показанно, что данный алгоритм демонстрирует достаточно хорошую производительность. Кроме того, основанный на матрицах CFPQ-алгоритм в дальнейшем стал основой для первой полноценной поддержки контекстно-свободных запросов в результате расширения графовой базы данных RedisGraph [14].

Матричное представление не единственный возможный способ решения такого рода задач. Более того, результаты исследований показали, что базовые алгоритма распознавания строк по заданной грамматике могут быть ествественным путем обобщены на входные данные в виде графа [8]. Например, существуют описания того, как проблема поиска путей с контекстно-свободными ограничениями может быть решена с помощью использования обобщенного LL-парсера [12, 9] или обобщенного LR-парсера [18, 20]. Более того, такой подход позволяет получать информацию не только о достижимостях в графе, но и о самих путях [11].

Так, в рамках предыдущих работ [22] была получена реализация алгоритма для выполнения контекстно-свободных запросов в графовой базе данных Neo4j. Данный алгоритм является адаптацией классического алгоритма синтаксического анализа Generlized LL для выполнения контекстно-свободных запросов на графах. Важно отметить, что полученный алгоритм поддерживает весь класс контекстно-свободных языков. Модифицированный GLL, так же как и оригинальный алгоритм, возвращает информацию не только о достижимостях между вершинами, но и информацию для построения самих путей. Несмотря на то, что проведенное экспериментальное исследование показало практическую эффективность полученного решения при работе с контекстно-свободными грамматиками, тем не менее данная реализация не позволяет напрямую работать наиболее распространненым на сегодняший день способом задания ограничений — с регулярными выражениями. Для их обработки сначала необходимо преобразование грамматики, что ведет к увеличению количества продукций в ней и, как следствие, ухудшению производительности алгоритма GLL. Обойти данную проблему может помочь выражение грамматик в виде реккурсивных автоматов,

которые одинаково оптимально поддерживают как регулярные, так и контекстно-свободные грамматики.

Таким образом, целью данной работы является предоставление реализации GLL алгоритма, основанного на рекурсивном автомате, для обработки входных данных в виде графа как для решения задачи достижимости, так и для поиска путей. Далее планируется проведение экспериментального исследования с целью выявления значимого повышения эффективности по сравнению с альтернативными решениями.

1 Постановка задачи

Целью данной работы является реализация и экспериментальное исследование производительности GLL-парсера, основанного на рекурсивном автомате.

Для достижения данной цели были поставлены следующие задачи.

- Реализация классического алгоритма GLL.
- Модификация алгоритма GLL для поддержки представления грамматики в виде рекурсивного автомата.
- Расширение модифицированного алгоритма GLL на входные данные в виде графа.
- Проведение экспериментального исследования и сравнение с существующими решениями.

2 Обзор

Данный раздел содержит ряд базовых определений из теории формальных языков, а далее — краткое описание алгоритмов поиска путей с контекстно-свободными ограничениями.

2.1 Базовые определения из теории формальных языков

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

Определение 2.1. *Контекстно-свободная грамматика* — это четверка $\mathbb{G} = \langle N, \Sigma, P, S \rangle$, где

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

Важную роль в данной работе играет понятие *выводимости слова в контекстно-свободной грамматике*. Необходимые определения приведены ниже.

Определение 2.2. *Шаг вывода в контекстно-свободной грамматике* $\mathbb{G} = \langle N, \Sigma, P, S \rangle$ — это переход $\gamma A \beta \Rightarrow \gamma \alpha \beta$, где $\gamma, \beta \in \{\Sigma \cup N\}^*$ и $(A \to \alpha) \in P$

Определение 2.3. Слово $w \in \Sigma^*$ *выводимо в контекстно-свободной грамматике* $\mathbb{G} = \langle N, \Sigma, P, S \rangle$, если существует последовательность шагов вывода в грамматике $\mathbb{G}$: $S \Rightarrow \beta_0 \Rightarrow \beta_1 \Rightarrow \cdots \Rightarrow \beta_{n-1} \Rightarrow w$. Будем обозначать это как $S \overset{*}{\Rightarrow} w$.

Определение 2.4. *Язык, задаваемый контекстно-свободной грамматикой* $\mathbb{G} = \langle N, \Sigma, P, S \rangle$ — это множество слов, выводимых в грамматике

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

Наравне с контекстно-свободными грамматиками в этой работе также используются рекурсивные автоматы, поэтому введем соответствующие определения.

Определение 2.5. *Детерминированный конечный автомат* — это пятерка $\langle Q, \Sigma, \sigma, Q_0, Q_F \rangle$:

- Q — конечное множество состояний;
- Σ — конечное множество входных символов;
- $\sigma : Q \times \Sigma \to Q$ — функция переходов, аргументами которой являются текущее состояние и входной символ, а значением — новое состояние;
- Q_0 — начальное состояние, $Q_0 \in Q$;
- Q_F — множество допускающих состояний, $Q_F \subseteq Q$.

Рекурсивный автомат является обобщением детерминированного конечного автомата и позволяет задавать те же языки, что задаются контекстно-свободными грамматиками.

Определение 2.6. *Рекурсивный автомат* — это семерка

$$\mathbb{A} = \langle Q, \Sigma, N, \sigma, Q_S, Q_F, S \rangle$$

- Q — конечное множество состояний, каждое состояние помечено нетерминалом;
- N — конечное множество нетерминальных символов;
- Σ — конечное множество терминальных символов, $\Sigma \cap N = \varnothing$;

- $\sigma : Q \times (\Sigma \cup N) \to Q$ — функция переходов, аргументами которой являются текущее состояние и символ из $\Sigma \cup N$, а значением — новое состояние;
- Q_S — множество начальных состояний, $Q_S \subseteq Q$;
- Q_F — множество допускающих состояний, $Q_F \subseteq Q$;
- S — стартовый нетерминал, $S \in N$.

Рекурсивный автомат отличается от детерминированного конечного автомата тем, что:

- вместо одного конечного автомата, рекурсивный автомат может состоять из нескольких конечных автоматов, каждый из который помечен своим уникальным нетерминалом;
- переходы по терминальным символам осуществляются также, как и в детерминированном конечном автомате, однако переходы по нетерминальным символам осуществляются тогда и только тогда, когда рекурсивный спуск в конечный автомат, стартовое состояние которого помечено нетерминальным символом перехода, завершается в допускающем состоянии соответствующего автомата.

Пример 2.1. Рассмотрим язык $\mathbb{L}_0 = \{a^n b^n \mid n \in \mathbb{N}\}$. Язык $\mathbb{L}_0$ представляет собой конкатенацию n букв "a" и n букв "b", где n — натуральное число.

Этот язык может быть задан с помощью контекстно-свободной грамматики.

Пример 2.2. *Контекстно-свободная грамматика $\mathbb{G}_0$, задающая язык $\mathbb{L}_0$.*

$$S \to aSb$$

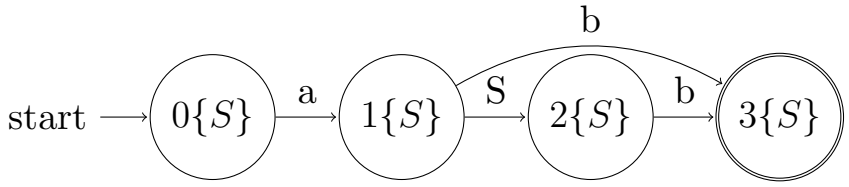
$$S \to ab$$

Пример 2.3. *Пример вывода слова $aaabbb \in \mathbb{L}_0$ в грамматике $\mathbb{G}_0$.*

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aaabbb$$

Этот же язык возможно задать с помощью рекурсивного автомата.

Пример 2.4. *Рекурсивный автомат* $\mathbb{A}_0$*, задающий язык* $\mathbb{L}_0$.



(1)

Пример 2.5. *Пример вывода слова* $aaabbb \in \mathbb{L}_0$ *в рекурсивном автомате* $\mathbb{A}_0$.

$$0 \overset{a}{\Rightarrow} 1 \underset{\text{rec}\downarrow}{\overset{S}{\Longrightarrow}} 0 \overset{a}{\Rightarrow} 1 \underset{\text{rec}\downarrow}{\overset{S}{\Longrightarrow}} 0 \overset{a}{\Rightarrow} 1 \overset{b}{\Rightarrow} 3 \underset{\text{rec}\uparrow}{\overset{S}{\Longrightarrow}} 2 \overset{b}{\Rightarrow} 3 \underset{\text{rec}\uparrow}{\overset{S}{\Longrightarrow}} 2 \overset{b}{\Rightarrow} 3$$

Здесь rec $\downarrow$ обозначает рекурсивный спуск в рекурсивном автомате, а rec $\uparrow$ возврат из рекурсивного спуска.

2.2 Базовые определения из теории графов

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

Определение 2.7. *Помеченный ориентированный граф* — это тройка $\mathbb{D} = \langle V, E, T \rangle$, где

- V — конечное множество вершин, не умаляя общности пронумеруем вершины натуральными числами от 0 до $|V| - 1$;
- T — конечное множество меток на ребрах;
- $E \subseteq V \times T \times V$ — конечное множество ребер.

Определение 2.8. *Путь* π *в графе* $\mathbb{D} = \langle V, E, T \rangle$ — это конечная последовательность ребер $(e_0, e_1, ..., e_{n-1})$, где $\forall\ j,\ 0 \leq j \leq n-1 : e_j = (v_j, t_j, v_{j+1}) \in E$. Определим множество всех путей в графе $\mathbb{D}$ как $\pi(\mathbb{D})$.

Определение 2.9. *Слово, соответствующее пути* $\pi = (e_0, e_1, ..., e_{n-1})$, определим как последовательную конкатенацию меток ребер пути $l(\pi) = t_0 t_1 \dots t_{n-1}$.

2.3 Задачи поиска путей с контекстно-свободными ограничениями

Теперь мы можем приступить к формальным определениям задач поиска путей с контекстно-свободными ограничениями.

Пусть даны:

- контекстно-свободная грамматика $\mathbb{G} = \langle N, \Sigma, P, S \rangle$;
- помеченный ориентированный граф $\mathbb{D} = \langle V, E, T \rangle$;
- множество стартовых вершин $V_S \subseteq V$ и множество финальных вершин $V_F \subseteq V$.

В введенных обозначениях могут быть сформулированы следующие задачи.

- **Задача поиска путей в графе с контекстно-свободными ограничениями** состоит в нахождении всех путей в графе таких что $l(\pi) \in L(\mathbb{G})$ и $v_0 \in V_S,\ v_n \in V_F$.
- **Задача поиска достижимостей в графе с контекстно-свободными ограничениями** заключается в поиске множества пар вершин, для которых существует такой путь с началом и концом в этих вершинах, что слово, составленное из меток рёбер пути, принадлежит заданному языку: $\{(v_i, v_j) \mid \exists\ l(\pi) :\ \ l(\pi) \in L(\mathbb{G})$ и $v_0 \in V_S,\ v_n \in V_F\}$.

2.4 Обобщенный LL алгоритм

Одной из наиболее популярных техник синтаксического разбора является LL(k)-парсер, который представляет собой алгоритм нисходящего анализа с предпросмотром. Это означает, что решение о том, какая из продукций контекстно-свободной грамматики должна быть применена, базируется на предпросмотре k следующих от текущего символов. Для выбора правильной продукции поддерживается специальная таблица, в которой хранится информация для разбора текущего символа.

Однако, этот алгоритм может быть применен лишь к небольшому подмножеству контекстно-свободных грамматик. Например, данный алгоритм неприменим к неоднозначным грамматикам или грамматикам с непосредственной левой рекурсией.

Алгоритмы нисходящего анализа относительно проще реализовывать и отлаживать, чем алгоритмы восходящего анализа, так как они полностью соотносятся со структурой грамматики. По этой причине для того, чтобы поддержать весь класс контекстно-свободных грамматик, был предложен обобщенный вариант LL(k) алгоритма — GLL (Generalized LL) [19]. В случае LL(k) алгоритма может возникнуть такая ситуация, что невозможно определить, какая продукция должна быть выбрана в текущем состоянии синтаксического разбора. Для решения данной проблемы в GLL вместо выбора одной продукции выбираются все допустимые продукции. Такой подход позволяет во время работы алгоритма рассмотреть все возможные переходы из одного состояния разбора в другое.

Процесс работы GLL можно рассматривать как процесс обхода грамматики, управляемый входной строкой. В каждый момент времени работы алгоритма GLL поддерживается дескриптор, состоящий из следующих объектов: $\langle A \to \alpha, c_\alpha \rangle$ — *позиция в грамматике*, c_i — *позиция во входной строке*, c_u — *текущая вершина GSS*, c_N — *текущая вершина SPPF*.

Определение 2.10. *Позиция в грамматике* — это пара $\langle A \to \alpha, c_\alpha \rangle$, содержащая текущую продукцию в грамматике $A \to \alpha$ и индекс текущего символа в продукции соответственно. Наряду с обозначением $\langle A \to \alpha, c_\alpha \rangle$ примем обозначение $A \to \alpha \cdot X\beta$, где c_α соответствует индексу символа X, который может быть как терминальным, так и нетерминальным.

Определение 2.11. *Позиция во входной строке* c_i — это индекс текущего символа во входной строке.

Определение 2.12. *GSS* или *графо-структурированный стек* — граф, предназначенный для замены стека из LL(k)-парсера, вершины которо-

го представляют пары $\langle \mathbb{L}, c_i \rangle$, что соответствует позиции в грамматике и позиции во входной строке на момент создания вершины, то есть некоторому состоянию разбора, а ребра обозначают рекурсивные переходы по грамматике и помечены соответствующими SPPF вершинами.

Определение 2.13. *SPPF* — сжатое представление леса разбора, которое содержит в себе все деревья вывода входной строки.

Четверка $\langle \langle A \to \alpha, c_\alpha \rangle, c_u, c_i, c_N \rangle$ — называется дескриптором. В ходе работы GLL алгоритма поддерживается два множества:

- $\mathbb{R}$ — множество всех еще необработанных дескрипторов, представленное в виде очереди;
- $\mathbb{U}$ — множество всех уже обработанных дескрипторов.

Изначально $\mathbb{R} = \{\langle \langle S, 0 \rangle, c_u^\varnothing, 0, c_N^\varnothing \rangle\}$ и $\mathbb{U} = \varnothing$, где $c_u^\varnothing$ — специальная стартовая GSS вершина, $c_N^\varnothing$ — пустое SPPF дерево. Когда GLL алгоритм встречает терминальный символ на текущей позиции в грамматике $\langle A \to \alpha, c_\alpha \rangle$, если он совпадает с текущим символом входной строки, то в $\mathbb{R}$ добавляется дескриптор $\langle \langle A \to \alpha, c_\alpha + 1 \rangle, c_u, c_i + 1, c_N^{new} \rangle$, где c_N^{new} является SPPF деревом разбора подстроки $0 \dots c_i + 1$, иначе, если символ на текущей позиции в грамматике не совпадает с текущим символом входной строки, то осуществляеся переход к следующему дескриптору в очереди $\mathbb{R}$. Когда GLL алгоритм встречает нетерминальный символ на текущей позиции в грамматике $A \to \alpha \cdot X\beta$, создается вершина GSS c_u^{new}, содержащая текущее состояние разбора, и для каждой продукции $(X \to \gamma) \in P$ в $\mathbb{R}$ добавляются дескрипторы вида $\langle \langle X \to \gamma, 0 \rangle, c_u^{new}, c_i, c_N^\varnothing \rangle$, которых еще нет в $\mathbb{U}$. GLL алгоритм завершается тогда, когда $\mathbb{R} = \varnothing$.

2.5 Алгоритм выполнения контекстно-свободных запросов, базирующийся на GLL

Как было сказано выше, алгоритмы на основе нисходящего анализа могут быть использованы для решения задачи поиска путей с

контекстно-свободными ограничениями. Более того, GLL алгоритм довольно естественным путем обобщается на входные данные в виде графов [9]: вместо того, чтобы позициями входа считать индексы линейного входа, теперь будем считать ими вершины графа.

Для сохранения корректности работы алгоритма вносятся следующие изменения.

- Запрос теперь представляет собой тройку: множество стартовых вершин, множество финальных вершин и грамматику.
- *Позиция во входной строке* заменяется на вершину графа.
- Начальное множество дескрипторов инициализируется дескрипторами, каждый из которых содержит стартовую вершину запроса.
- На шаге перехода в грамматике обрабатываются все переходы по исходящим ребрам текущей вершины графа;
- Если разбор завершен, необходимо проверить принадлежность вершины, на которой алгоритм остановился, к множеству финальных вершин графа.

Так же, как и в случае стандартного GLL алгоритма, данная модификация способна обрабатывать произвольные контекстно-свободные грамматики.

3 Реализация алгоритма

Прежде чем приступать к реализации модификации приведем пример, иллюстрирующий мотивацию предлагаемой модификации.

3.1 Мотивация

В классическом GLL алгоритме дескриптор содержит в себе *позицию в грамматике* — текущую обрабатываемую продукцию и позицию в этой продукции, обозначающаю какие элементы продукции уже были обработаны. Таким образом для каждой продукции грамматики в GLL алгоритме неизбежно создается как минимум столько дескрипторов, сколько в этой продукции символов.

Пример 3.1. Рассмотрим простой язык, состоящий из нуля и более повторений буквы a: $\mathbb{L}_1 = \{a\}^*$.

Пример 3.2. Контекстно-свободная грамматика $\mathbb{G}_1$, задающая язык $\mathbb{L}_1$.

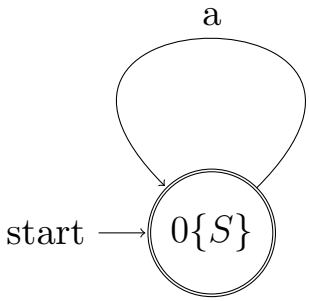
$$S \longrightarrow aS$$

$$S \longrightarrow \varepsilon$$

Для такой грамматики в ходе работы GLL алгоритма будет создано целых четыре *позиции в грамматике*, и, соответственно, не меньше четырех дескрипторов.

Однако этот же язык можно выразить с помощью рекурсивного автомата всего на одном состоянии и одном переходе, а для такого автомата количество созданных GLL алгоритмом дескрипторов будет гораздо меньше.

Пример 3.3. Рекурсивный автомат $\mathbb{A}_1$, задающий язык $\mathbb{L}_1$.



(2)

Так как производительность GLL алгоритма напрямую зависит от количества создаваемых дескрипторов, возникает предположение о модификации классического GLL алгоритма путем замены контекстно-свободной грамматики на рекурсивный автомат.

3.2 Особенности реализации

Предлагаемая в данной работе модификация GLL алгоритма основана на замене в дескрипторе *позиции в грамматике* на состояние рекурсивного автомата, то есть в четверке $\langle\langle A \to \alpha, c_\alpha\rangle, c_u, c_i, c_N\rangle$ пара $\langle A \to \alpha, c_\alpha\rangle$ заменяется на c_R — текущее состояние рекурсивного автомата. Соответственно переход в контекстно-свободной грамматике заменяется на переход в рекурсивном автомате.

Данная модификация, как и классический GLL алгоритм, может быть обобщена на графы — на шаге перехода в рекурсивном автомате обрабатываются все переходы из текущего состояния рекурсивного автомата по всем исходящим ребрам текущей вершины графа.

Одной из недавних удачных реализаций GLL алгоритма является библиотека GLL4Graph[1], написанная на языке программирования Java. Это означает, что виртуальная машина Java, она же JVM, является подходящей для эффективной реализации такого рода алгоритмов. С другой стороны, опираясь на то, что данная работа является возможной основой для дальнейших расширений и оптимизаций, не менее важным было не только разработать оптимальную для этих целей архитекту-

[1]Github репозиторий библиотеки: https://github.com/FormalLanguageConstrainedPathQuerying/GLL4Graph. Accessed: 05/04/2023

ру, но и сделать сам код реализации максимально лаконичным и понятным. По сравнению с языком программирования Java, программа, написанная на языке программирования Kotlin имеет более читаемый и структурно точный код, что облегчает не только ее понимание, но и дальнейшую поддержку. Поэтому в качестве языка программирования для реализации модификации GLL алгоритма был выбран язык Kotlin.

3.3 Архитектура

Прежде, чем приступать к модификации GLL алгоритма, необходимо реализовать его классический вариант. В связи с этим возникает вопрос о разработке подходящей архитектуры для проекта в целом. Поэтому были разработаны следующие требования.

- Для удобства модификации и дальнейшей поддержки кода каждая реализация GLL алгоритма должна находиться в отдельном модуле и не зависеть от других реализаций.
- Кроме GLL алгоритмов, с входными данными, представленными в виде строки, необходимо реализовать GLL алгоритмы, с входными данными, представленными в виде графа.
- Помимо решения задачи поиска путей в графе с контекстно-свободными ограничениями необходимо поддержать решение задачи поиска достижимостей в графе с контекстно-свободными ограничениями.
- Наконец, необходимо предоставить некоторый удобный пользователю интерфейс с возможностью выбора конкретного сценария для GLL алгоритма.

Для удовлетворения поставленным требованиям была разработана следующая архитектура.

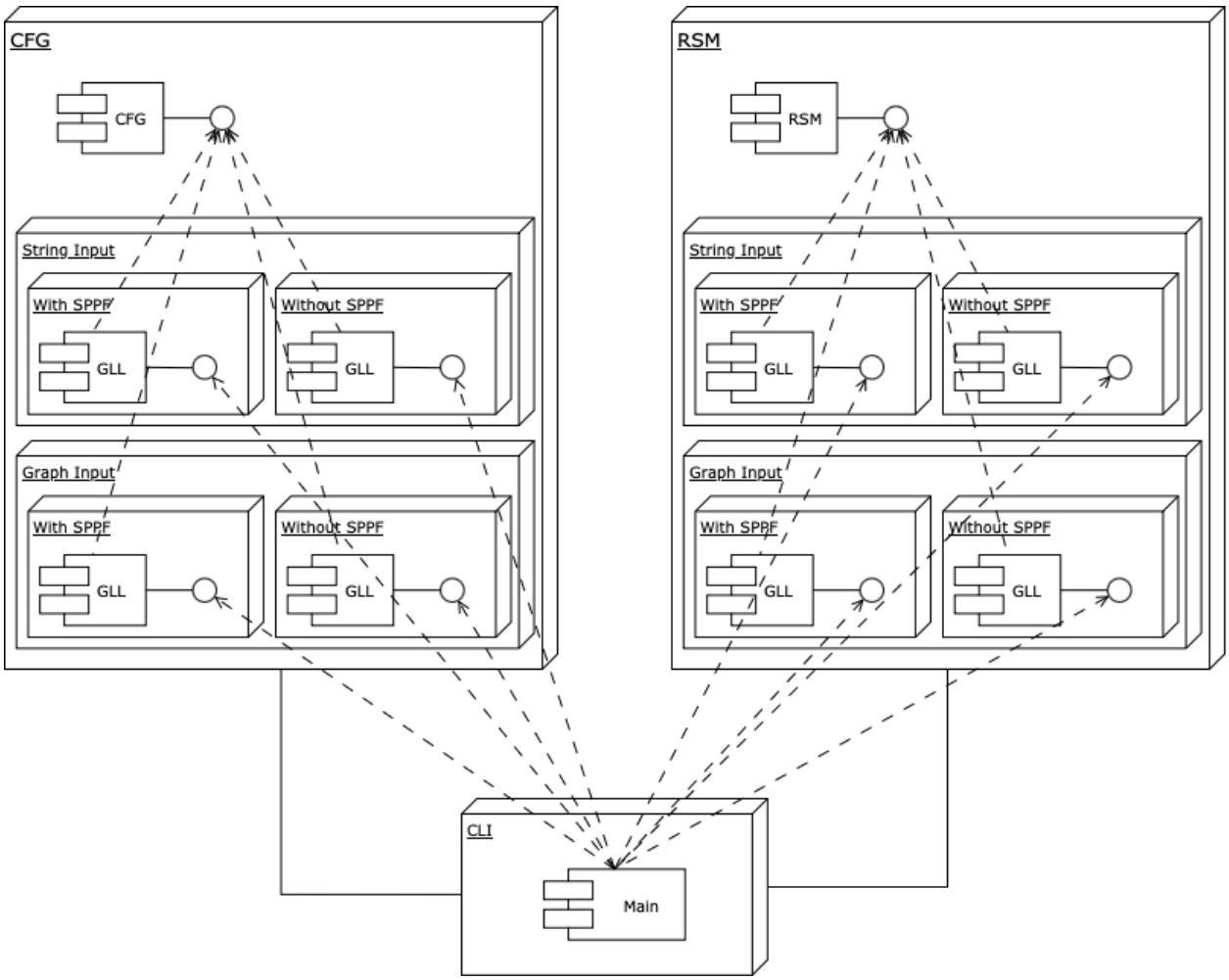


Рис. 1: Архитектура проекта

Разработанный проект имеет блочно-модульную архитектуру. Диаграмма включает в себя три основных блока — снизу блок, содеражащий модуль, реализующий интерфейс командой строки, который был отделен от модулей реализаций GLL, сверху слева блок, представляющий собой реализации GLL алгоритма на основе контекстно-свободных грамматик, имеющий общий модуль представления контекстно-свободной грамматики для всех GLL алгоритмов в блоке, сверху справа — блок, представляющий собой реализации GLL алгоритма на основе рекурсивного автомата, имеющий общий модуль представления рекурсивного автомата для всех GLL алгоритмов в блоке. Для обоих типов реализации были инкапсулированы в отдельные блоки сценарии выполнения алгоритма для входных данных в виде строки и для входных данных в виде графа. Внутри последних в свою очередь были инкапсулированы

сценарии с построением дерева разбора SPPF и без построения дерева разбора.

Данная архитектура позволяет не только максимально гибко задавать параметры для запуска алгоритма через интерфейс командой строки, но и облегчает понимание и поддержку кода, так как реализаци GLL алгоритмов имеют строгую иерархию и независимы между собой.

4 Экспериментальное исследование

Для оценки производительности предложенного решения было проведено экспериментальное исследование на реальных графах и запросах. В данном разделе приведены и описаны полученные результаты.

4.1 Оборудование

Все эксперименты были запущены на сервере со следующими характеристиками.

- Операционная система Ubuntu 18.04.
- Процессор Intel Core i7-6700 CPU, 3.40GHz, 4 потока (hyper-threading выключен).
- DDR4 64Gb RAM.
- OpenJDK 64-Bit виртуальная машина (build 25.362-b09, mixed mode). JVM сконфигурирована использовать 60Gb памяти кучи.

4.2 Эсперименталные данные

Экспериментальное исследование проводилось на графах, содержащихся в библиотеке CFPQ_Data[2]. Был выбран набор графов, относящихся к RDF анализу (класс RDF), как достаточно известный и зарекомендовавший себя со временем набор графов для исследования решающих задачу поиска достижимостей в графе с контекстно-свободными ограничениями. Подробное описание графов (количество вершин, ребер, а также количество ребер с метками, используемых в запросе) приведено в таблице 1.

Для графов, из класса RDF были использованы те же запросы, что и в других работах по разработке алгоритмов решающих задачу поиска достижимостей в графе с контекстно-свободными ограничениями:

[2]CFPQ_Data — это публичный набор графов и грамматик для CFPQ-алгоритмов. GitHub репозиторий: https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_Data. Accessed: 05/04/2023.

Название графа	$\|V\|$	$\|E\|$	#subClassOf	#type	#broaderTransitive
skos	144	252	1	70	0
generations	129	273	0	78	0
travel	131	277	30	90	0
univ	179	293	36	84	0
atom	291	425	122	138	0
biomedical	341	459	122	130	0
foaf	256	631	10	174	0
people	337	640	33	161	0
funding	778	1086	90	304	0
wine	733	1839	126	485	0
pizza	671	1980	259	365	0
core	1323	2752	178	706	0
pathways	6238	12363	3117	3118	0
enzyme	48815	86543	8163	14989	8156
eclass	239111	360248	90962	72517	0
go_hierarchy	45007	490109	490109	0	0
go	582929	1437437	94514	226481	0
geospecies	450609	2201532	0	89062	20867
taxonomy	5728398	14922125	2112637	2508635	0

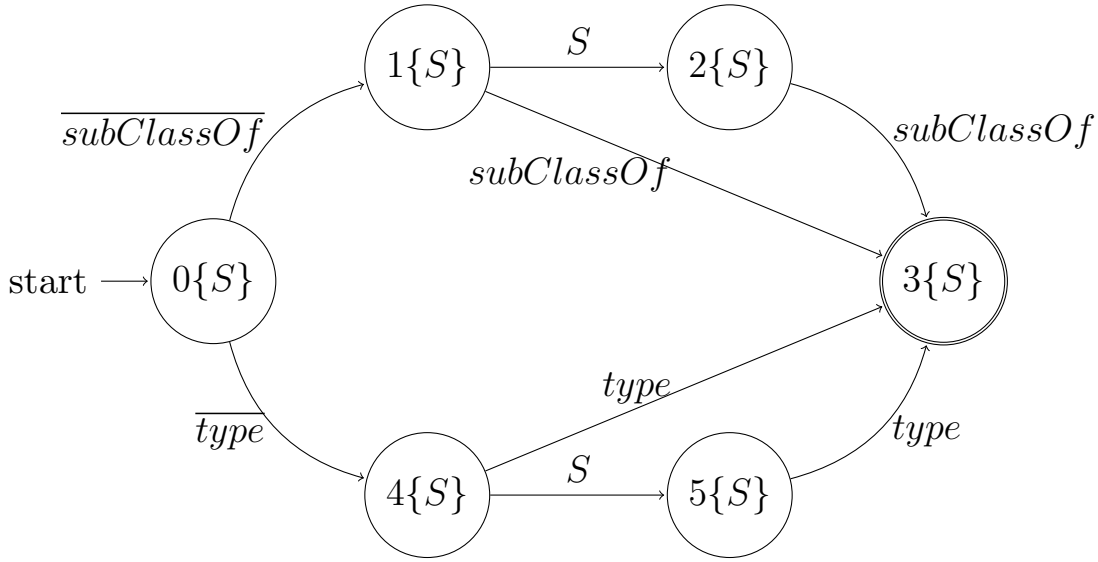
Таблица 1: Графы класса RDF: количество вершин, ребер и ребер с определенными метками

G_1 (3), G_2 (5) и Geo (7). Запросы выражены как контекстно-свободные грамматики, где S — стартовый нетерминал, *subClassOf, type, broaderTransitive*, $\overline{subClassOf}$, $\overline{type}$, $\overline{broaderTransitive}$ — терминальные символы. Здесь $\overline{x}$ обозначает обратное ребро (v_j, x, v_i) для ребра (v_i, x, v_j) в графе.

Пример 4.1. Контекстно-свободная грамматика G_1.

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

Пример 4.2. Рекурсивный автомат A_1, соответствующий контекстно-свободной грамматике G_1.

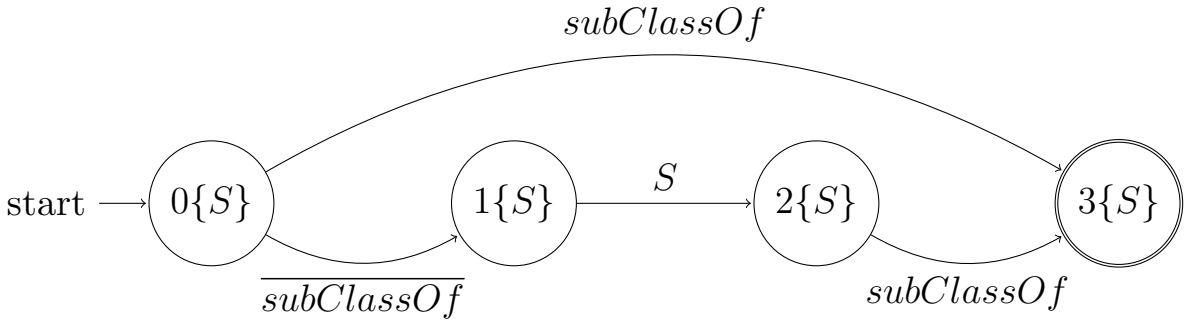


(4)

Пример 4.3. Контекстно-свободная грамматика G_2.

$$S \to \overline{subClassOf}\ \ S\ subClassOf \mid subClassOf \tag{5}$$

Пример 4.4. Рекурсивный автомат A_2, соответствующий контекстно-свободной грамматике G_2.

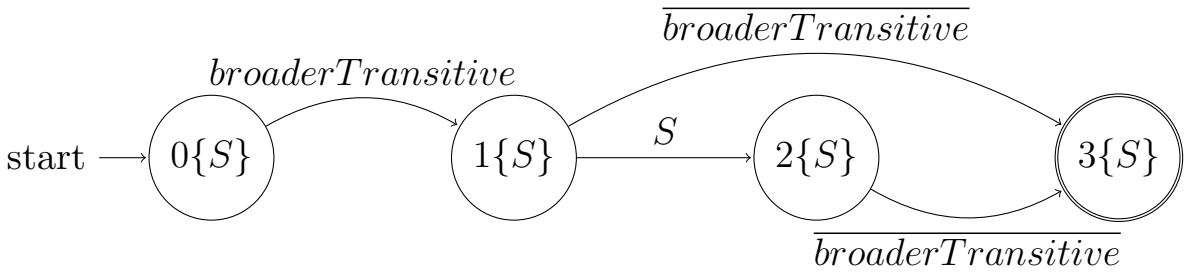


(6)

Пример 4.5. Контекстно-свободная грамматика Geo.

$$\begin{aligned} S \to & broaderTransitive\ \ S\ \overline{broaderTransitive} \\ & \mid broaderTransitive\ \ \overline{broaderTransitive} \end{aligned} \tag{7}$$

Пример 4.6. Рекурсивный автомат A_{geo}, соответствующий контекстно-свободной грамматике Geo.



(8)

Кроме того, для исследования различий в эффективности обработки регулярных выражений классическим и модифицированным GLL алгоритмами, были выбраны следующие регулярные выражения, построенные по шаблонам Q_1, Q_2, Q_4^2 и Q_6 из работы [15].

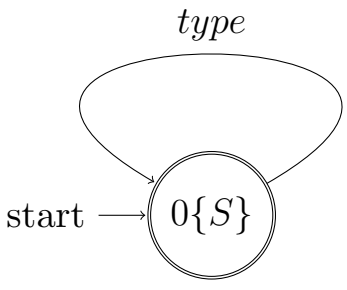
Пример 4.7. Регулярное выражение R_1 для класса графов RDF, построенное по шаблону Q_1 на терминальном символе *type*.

$$type^* \tag{9}$$

Пример 4.8. Контекстно-свободная грамматика G_{R_1}, соответствующая регулярному выражению (9).

$$\begin{aligned} S &\to \varepsilon \\ S &\to type\ S \end{aligned} \tag{10}$$

Пример 4.9. Рекурсивный автомат A_{R_1}, соответствующий регулярному выражению (9).



(11)

Пример 4.10. Регулярное выражение R_2 для класса графов RDF, по-

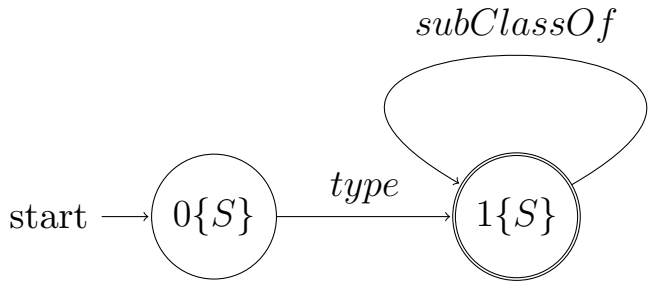
строенное по шаблону Q_2 на терминальных символах $type$ и $subClassOf$ соответственно.

$$type\ subClassOf^* \tag{12}$$

Пример 4.11. Контекстно-свободная грамматика G_{R_2}, соответствующая регулярному выражению (12).

$$\begin{aligned} S &\to type\ C \\ C &\to \varepsilon \\ C &\to subClassOf\ C \end{aligned} \tag{13}$$

Пример 4.12. Рекурсивный автомат A_{R_2}, соответствующий регулярному выражению (12).



(14)

Пример 4.13. Регулярное выражение R_3 для класса графов RDF, построенное по шаблону Q_4^2 на терминальных символах $type$ и $subClassOf$ соответственно.

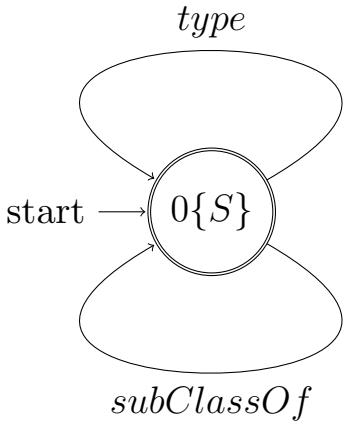
$$(type \mid subClassOf)^* \tag{15}$$

Пример 4.14. Контекстно-свободная грамматика G_{R_3}, соответствующая регулярному выражению (15).

$$\begin{aligned} S &\to \varepsilon \\ S &\to type\ S \\ S &\to subClassOf\ S \end{aligned} \tag{16}$$

Пример 4.15. Рекурсивный автомат A_{R_3}, соответствующий регуляр-

ному выражению (15).



(17)

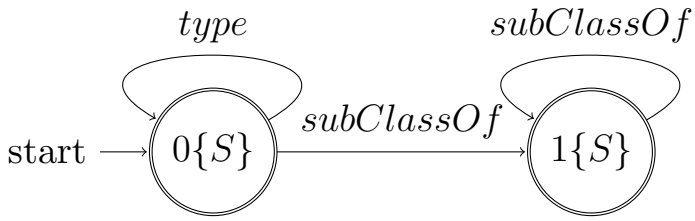
Пример 4.16. Регулярное выражение R_4 для класса графов RDF, построенное по шаблону Q_6на терминальных символах *type* и $subClassOf$ соответственно.

$$type^*\ subClassOf^* \tag{18}$$

Пример 4.17. Контекстно-свободная грамматика G_{R_4}, соответствующая регулярному выражению (18).

$$\begin{aligned} &S \to A\ B \\ &A \to \varepsilon \\ &A \to type\ A \\ &B \to \varepsilon \\ &B \to subClassOf\ B \end{aligned} \tag{19}$$

Пример 4.18. Рекурсивный автомат A_{R_4}, соответствующий регулярному выражению (18).



(20)

4.3 Постановка экспериментов

Экспериментальное исследование проводилось следующий образом.

- Замерялось время работы классического и модифицированного GLL алгоритмов на задачах поиска путей в графе с контекстно-свободными ограничениями, поиска достижимостей в графе с контекстно-свободными ограничениями и поиска достижимостей в графе с регулярными ограничениями.

- В качестве стартового и в качестве финального множеств запроса были расмотрены все вершины графа.

- В качестве контекстно-свободных ограничений были рассмотрены контекстно-свободные грамматики G_1 (3), G_2 (5), Geo (7) и соответствующие им рекурсивные автоматы A_1 (4), A_2 (6), и A_{geo} (8).

- В качестве регулярных ограничений были рассмотрены регулярные выражения R_1 (9), R_2 (12), R_3 (15) и R_4 (18), соответствующие им контекстно-свободные грамматики G_{R_1} (10), G_{R_2} (13), G_{R_3} (16) и G_{R_4} (19) представляли запросы для классического GLL алгоритма, а соответствующие им рекурсивные автоматы A_{R_1} (11), A_{R_2} (14), A_{R_3} (17) и A_{R_4} (20) представляли запросы для модифицированного GLL алгоритма.

- Для каждой четверки (граф, грамматика, GLL алгоритм, задача) проводилось 20 замеров времени работы в наносекундах.

4.4 Результаты экспериментов

Для удобства исследования полученные результаты были оформлены в виде таблиц, представленных ниже. Результаты приведены с точностью до последней значащей цифры, выбранной с учетом стандартного отклонения для каждой выборки результатов.

Ниже представлены результаты работы классического GLL алгоритма в сравнении с модифицированным алгоритмом для сценария поиска всех путей и сценария поиска всех достижимостей на графах класса RDF с контекстно-свободными ограничениями.

Название графа	Время в секундах					
	G1		G2		Geo	
	CFG	RSM	CFG	RSM	CFG	RSM
skos	0.00021 ± 0.00004	0.0001 ± 0.000005	0.000038 ± 0.000006	0.000023 ± 0.000004	0.000037 ± 0.000009	0.000021 ± 0.000005
generations	0.000713 ± 0.001008	0.000226 ± 0.000005	0.000105 ± 0.000018	0.000086 ± 0.000008	0.00009 ± 0.000008	0.000064 ± 0.000006
travel	0.00079 ± 0.00004	0.000425 ± 0.000007	0.00028 ± 0.00002	0.000255 ± 0.000006	0.000092 ± 0.000006	0.000066 ± 0.000011
univ	0.0009 ± 0.0003	0.000416 ± 0.000010	0.00036 ± 0.00003	0.000273 ± 0.000019	0.00013 ± 0.00002	0.00009 ± 0.000010
atom	0.00058 ± 0.00007	0.000261 ± 0.000012	0.00028 ± 0.00002	0.000211 ± 0.000010	0.000078 ± 0.000019	0.000047 ± 0.000012
biomedical	0.00181 ± 0.00018	0.001 ± 0.0002	0.0012 ± 0.0004	0.00082 ± 0.00006	0.000232 ± 0.000012	0.000161 ± 0.000014
foaf	0.0015 ± 0.0006	0.00074 ± 0.00004	0.00029 ± 0.00003	0.0002 ± 0.00002	0.00021 ± 0.00004	0.00013 ± 0.00002
people	0.00133 ± 0.00003	0.0007 ± 0.00003	0.00047 ± 0.00005	0.0004 ± 0.00002	0.00023 ± 0.00002	0.00016 ± 0.00002
funding	0.00118 ± 0.00009	0.0005 ± 0.0002	0.0004 ± 0.00005	0.00026 ± 0.00002	0.0004 ± 0.0003	0.00013 ± 0.00003
wine	0.0028 ± 0.0007	0.0031 ± 0.0002	0.0019 ± 0.0003	0.00156 ± 0.00016	0.00053 ± 0.00002	0.00037 ± 0.00004
pizza	0.0065 ± 0.0005	0.0053 ± 0.0004	0.004 ± 0.0006	0.0031 ± 0.00018	0.0005 ± 0.00014	0.00033 ± 0.00002
core	0.0094 ± 0.0009	0.0055 ± 0.0011	0.0034 ± 0.0006	0.0028 ± 0.0004	0.0015 ± 0.0006	0.0012 ± 0.0003
pathways	0.04 ± 0.004	0.02 ± 0.004	0.03 ± 0.009	0.01104 ± 0.00103	0.00255 ± 0.00013	0.00153 ± 0.00016
enzyme	0.16 ± 0.03	0.076 ± 0.019	0.062 ± 0.008	0.049 ± 0.009	54.7 ± 1.4	53.7 ± 1.2
eclass	1.8 ± 0.3	0.91 ± 0.18	0.9 ± 0.17	0.68 ± 0.12	0.23 ± 0.18	0.13 ± 0.03
go_hierarchy	23.83 ± 1.03	19.9 ± 0.7	21.6 ± 0.8	19.4 ± 0.7	0.0296 ± 0.0014	0.016 ± 0.004
go	OOM	OOM	OOM	OOM	0.48 ± 0.06	0.27 ± 0.07
geospecies	OOM	OOM	OOM	OOM	OOM	OOM
taxonomy	OOM	OOM	OOM	OOM	7.7 ± 0.4	4.4 ± 0.5

Таблица 2: Результаты экспериментов на задаче поиска путей в графе с контекстно-свободными ограничениями

Название графа	Время в секундах					
	G1		G2		Geo	
	CFG	RSM	CFG	RSM	CFG	RSM
skos	0.00027 ± 0.00002	0.00005 ± 0.000002	0.000039 ± 0.000010	0.000027 ± 0.000011	0.000055 ± 0.000005	0.000024 ± 0.000010
generations	0.00047 ± 0.00004	0.00017 ± 0.000010	0.00013 ± 0.00002	0.000077 ± 0.000014	0.000082 ± 0.000001	0.000069 ± 0.000019
travel	0.00051 ± 0.00004	0.000267 ± 0.000014	0.00025 ± 0.00002	0.000149 ± 0.000018	0.000091 ± 0.000013	0.000069 ± 0.000010
univ	0.00062 ± 0.000021	0.0003 ± 0.00005	0.00026 ± 0.00004	0.00019 ± 0.00002	0.000133 ± 0.000015	0.000086 ± 0.000010
atom	0.000711 ± 0.000053	0.000133 ± 0.000002	0.000168 ± 0.000011	0.000116 ± 0.000018	0.000124 ± 0.000018	0.000038 ± 0.000005
biomedical	0.00124 ± 0.00007	0.00059 ± 0.00007	0.00081 ± 0.00008	0.00056 ± 0.00007	0.000238 ± 0.000017	0.000161 ± 0.000014
foaf	0.00082 ± 0.00004	0.00041 ± 0.00004	0.00026 ± 0.00003	0.000190 ± 0.000015	0.0002 ± 0.00004	0.000144 ± 0.000014
people	0.001 ± 0.0002	0.00051 ± 0.00008	0.00046 ± 0.00004	0.000261 ± 0.000013	0.00023 ± 0.00002	0.00015 ± 0.00002
funding	0.001551 ± 0.000109	0.000294 ± 0.000006	0.00028 ± 0.00002	0.000187 ± 0.000016	0.000244 ± 0.000019	0.00013 ± 0.00002
wine	0.0035 ± 0.0003	0.001639 ± 0.000103	0.0015 ± 0.00011	0.00081 ± 0.00007	0.00048 ± 0.00003	0.00035 ± 0.00003
pizza	0.0045 ± 0.0009	0.00051 ± 0.00008	0.00211 ± 0.00015	0.0015 ± 0.0002	0.00046 ± 0.00002	0.00031 ± 0.00002
core	0.0054 ± 0.0007	0.0031 ± 0.0009	0.0027 ± 0.0004	0.002 ± 0.0004	0.0014 ± 0.0004	0.0012 ± 0.0004
pathways	0.0169 ± 0.0016	0.0077 ± 0.0003	0.0088 ± 0.0007	0.00606 ± 0.00019	0.0026 ± 0.0002	0.0018 ± 0.0002
enzyme	0.107 ± 0.007	0.044 ± 0.008	0.049 ± 0.006	0.039 ± 0.014	8.6 ± 0.8	8.36 ± 1.02
eclass	0.943649 ± 0.14	0.43 ± 0.07	0.51 ± 0.11	0.39 ± 0.07	0.21 ± 0.18	0.13 ± 0.12
go_hierarchy	4.1 ± 0.6	3.0 ± 0.4	3.71 ± 0.15	3.5 ± 0.2	0.0272 ± 0.0015	0.0301 ± 0.0011
go	3.2 ± 0.3	1.86 ± 0.16	1.8 ± 0.3	1.49 ± 0.13	0.45 ± 0.07	0.28 ± 0.18
geospecies	0.97 ± 0.12	0.34 ± 0.04	0.36 ± 0.08	0.25 ± 0.05	279.2 ± 11.4	275.5 ± 4.6
taxonomy	31.2 ± 1.5	14.8 ± 0.6	17.7 ± 0.8	13.7 ± 0.6	6.9 ± 0.6	4.1 ± 0.6

Таблица 3: Результаты экспериментов на задаче поиска достижимостей в графе с контекстно-свободными ограничениями

Здесь OOM означает $Out\ of\ Memory$, то есть то, что для обработки графа оказалось недостаточно оперативной памяти.

Из полученных результатов можно сделать следующие выводы.

- Прежде всего, стоит заметить, что предоставленные модификации эффективны в контексте применимости алгоритма. В подавляющем большинстве случаев использование рекурсивного автомата дало положительный результат на времени работы GLL алгоритма.

- Как для классического GLL, так и для модифицированного большое значение имеет сруктура графа. Так, чем большее количество вершин содержит граф, тем сильнее на нем отражается улучшение во времени работы. Однако для относительно плотных графов (например, $go\_hierarchy$) ускорение заметно не так сильно.

- Значимым результатом является то, что модифицированный алгоритм работает быстрее не только на графах с большим количеством вершин, но также и на графах с запросами, возвращающими большое количество пар достижимостей (go, $taxonomy$). Так, например, на графе $taxonomy$ и запросе G_1 можно наблюдать прирост в производительности более, чем в два раза.

Ниже представлены результаты работы классического GLL алгоритма в сравнении с модифицированным алгоритмом для поиска всех достижимостей на графах класса RDF с регулярными ограничениями.

Название графа	Время в секундах							
	R1		R2		R3		R4	
	CFG	RSM	CFG	RSM	CFG	RSM	CFG	RSM
skos	0.000096 ± 0.000019	0.000160 ± 0.000019	0.00009 ± 0.00002	0.000041 ± 0.000014	0.000117 ± 0.000018	0.000062 ± 0.000012	0.0003 ± 0.00003	0.000056 ± 0.000013
generations	0.00027 ± 0.00013	0.000143 ± 0.000016	0.0004 ± 0.0005	0.000112 ± 0.000014	0.000276 ± 0.000005	0.000142 ± 0.000004	0.0003 ± 0.00004	0.000145 ± 0.000003
travel	0.00025 ± 0.00002	0.00015 ± 0.00002	0.00038 ± 0.00002	0.000133 ± 0.000016	0.000376 ± 0.000018	0.000180 ± 0.000013	0.00067 ± 0.00017	0.000176 ± 0.000013
univ	0.000331 ± 0.000018	0.0002 ± 0.0002	0.00039 ± 0.00006	0.000171 ± 0.000010	0.00047 ± 0.00002	0.00027 ± 0.00002	0.00094 ± 0.00007	0.00025 ± 0.00003
atom	0.00016 ± 0.00002	0.00028 ± 0.00002	0.000168 ± 0.000015	0.000076 ± 0.000010	0.00033 ± 0.00005	0.000120 ± 0.000016	0.00063 ± 0.00005	0.000132 ± 0.000017
biomedical	0.00065 ± 0.00006	0.00041 ± 0.00017	0.00067 ± 0.00005	0.00032 ± 0.00002	0.00117 ± 0.00008	0.00056 ± 0.00004	0.00205 ± 0.00015	0.00051 ± 0.00002
foaf	0.00058 ± 0.00005	0.000355 ± 0.000017	0.000667 ± 0.000105	0.00031 ± 0.00005	0.000734 ± 0.000100	0.00041 ± 0.00014	0.00136 ± 0.00004	0.00036 ± 0.00002
people	0.00061 ± 0.00002	0.000368 ± 0.000019	0.00084 ± 0.00016	0.0003 ± 0.00004	0.000799 ± 0.000019	0.00042 ± 0.00003	0.0016 ± 0.0008	0.00042 ± 0.00004
funding	0.00045 ± 0.00004	0.00083 ± 0.00009	0.00044 ± 0.00005	0.000210 ± 0.000019	0.0007 ± 0.00007	0.00036 ± 0.00004	0.001745 ± 0.000106	0.00035 ± 0.00005
wine	0.0018 ± 0.0007	0.00094 ± 0.00009	0.0025 ± 0.0002	0.000845 ± 0.000109	0.0023 ± 0.0002	0.0012 ± 0.00007	0.0029 ± 0.0004	0.0012 ± 0.0003
pizza	0.002 ± 0.0002	0.001 ± 0.0002	0.002 ± 0.0001	0.00079 ± 0.00006	0.0023 ± 0.0002	0.00111 ± 0.00003	0.0044 ± 0.0002	0.00109 ± 0.00005
core	0.005 ± 0.0003	0.003 ± 0.0004	0.0042 ± 0.0005	0.0019 ± 0.0002	0.0054 ± 0.0006	0.0031 ± 0.0005	0.012 ± 0.004	0.0031 ± 0.0003
pathways	0.0063 ± 0.0003	0.0068 ± 0.0107	0.0141 ± 0.0011	0.00631 ± 0.00015	0.0162 ± 0.0014	0.0078 ± 0.0004	0.06 ± 0.009	0.0082 ± 0.0005
enzyme	0.06 ± 0.011	0.0407 ± 0.0108	0.061 ± 0.011	0.032 ± 0.011	0.11 ± 0.02	0.056 ± 0.007	0.31 ± 0.08	0.0577 ± 0.0102
eclass	0.43 ± 0.11	0.28 ± 0.08	0.44 ± 0.11	0.19 ± 0.05	0.94 ± 0.17	0.50 ± 0.09	2.12 ± 0.27	0.46 ± 0.09
go_hierarchy	0.034 ± 0.002	0.039 ± 0.019	0.0375 ± 0.0109	0.0164 ± 0.0010	1.6 ± 0.2	0.42 ± 0.07	1.81 ± 0.09	0.4 ± 0.08
go	1.1 ± 0.2	0.7 ± 0.2	1.0 ± 0.2	0.461 ± 0.104	3.0 ± 0.3	1.8 ± 0.2	7.4 ± 2.7	1.6 ± 0.2
geospecies	0.64 ± 0.14	0.51 ± 0.14	0.95 ± 0.16	0.45 ± 0.06	0.9 ± 0.2	0.54 ± 0.13	3.1 ± 0.6	0.521 ± 0.109
taxonomy	OOM	OOM	OOM	OOM	OOM	OOM	OOM	OOM

Таблица 4: Результаты экспериментов на задаче поиска достижимостей в графе с регулярными ограничениями

Из полученных результатов можно сделать следующие выводы.

- Для небольших регулярных ограничений, в смысле мощности соответствующего языка, а именно, для R_1, время работы классического и модифицированного GLL алгоритмов примерно одинаково на всех графах.

- Чем регулярное ограничение больше, в смысле мощности соответствующего языка, тем эффективнее работает модифицированный GLL алгоритм, по сравнению с классическим.

Ниже представлены результаты работы модифицированного GLL алгоритма в сравнении с результатами работы библиотеки GLL4Graph на тех же графах и запросах для сценария поиска всех достижимостей с контекстно-свободными ограничениями.

Название графа	Время в секундах			
	G1		G2	
	RSM	GLL4Graph	RSM	GLL4Graph
skos	0.00005 ± 0.000002	0.005 ± 0.0001	0.000027 ± 0.000011	0.004 ± 0.0001
generations	0.00017 ± 0.000010	0.005 ± 0.0001	0.000077 ± 0.000014	0.004 ± 0.0001
travel	0.000267 ± 0.000014	0.006 ± 0.0001	0.000149 ± 0.000018	0.006 ± 0.0001
univ	0.0003 ± 0.00005	0.006 ± 0.0001	0.00019 ± 0.00002	0.005 ± 0.0001
atom	0.000133 ± 0.000002	0.008 ± 0.0001	0.000116 ± 0.000018	0.005 ± 0.0001
biomedical	0.00059 ± 0.00007	0.009 ± 0.0001	0.00056 ± 0.00007	0.007 ± 0.0001
foaf	0.00041 ± 0.00004	0.006 ± 0.0001	0.000190 ± 0.000015	0.005 ± 0.0001
people	0.00051 ± 0.00008	0.007 ± 0.0001	0.000261 ± 0.000013	0.007 ± 0.0001
funding	0.000294 ± 0.000006	0.012 ± 0.0001	0.000187 ± 0.000016	0.008 ± 0.0001
wine	0.001639 ± 0.000103	0.013 ± 0.0001	0.00081 ± 0.00007	0.01 ± 0.0001
pizza	0.00051 ± 0.00008	0.015 ± 0.0001	0.0015 ± 0.0002	0.01 ± 0.0001
core	0.0031 ± 0.0009	0.019 ± 0.0001	0.002 ± 0.0004	0.013 ± 0.0001
pathways	0.0077 ± 0.0003	0.06 ± 0.02	0.00606 ± 0.00019	0.04 ± 0.02
enzyme	0.044 ± 0.008	0.22 ± 0.01	0.039 ± 0.014	0.17 ± 0.01
eclass	0.43 ± 0.07	1.5 ± 0.03	0.39 ± 0.07	0.97 ± 0.03
go_hierarchy	3.0 ± 0.4	3.6 ± 0.2	3.5 ± 0.2	5.4 ± 0.2
go	1.86 ± 0.16	5.55 ± 0.08	1.49 ± 0.13	4.24 ± 0.08
geospecies	0.34 ± 0.04	2.89 ± 0.6	0.25 ± 0.05	2.65 ± 0.6
taxonomy	14.8 ± 0.6	45.4 ± 0.7	13.7 ± 0.6	36.069 ± 0.7

Таблица 5: Результаты сравнения с GLL4Graph на задаче поиска достижимостей в графе с контекстно-свободными ограничениями

Из полученных результатов можно сделать следующие выводы.

- Реализованная модификация оказывается в несколько раз эффективнее реализации GLL алгоритма в проекте GLL4Graph на большинстве графов.

- Исключением, опять же, является самый плотный из представленных графов — $go_h ierarchy$, на котором прирост производительности составил только около 20%.

- Это означает, что была получена не только эффективная реализация GLL алгоритма с использованием рекурсивного автомата, но и сам по себе классический GLL алгоритм также реализован достаточно оптимально.

- Получается, что реализованная модификация GLL алгоритма может быть весьма эффективна в практическом применении.

Заключение

В ходе работы были достигнуты следующие результаты.

- Реализован классический GLL алгоритм.
- Алгоритм GLL был модификацирован для поддержки представления грамматики в виде рекурсивного автомата.
- Реализовано расширение модификацированного GLL алгоритма на входные данные в виде графа.
- Проведено экспериментальное исследование, по результатам которого можно сделать выводы о том, что полученная модификация алгоритма GLL работает существенно эффективнее классического алгоритма GLL.

Можно выделить несколько дальнейших возможных направлений работы.

- Проведение экспериментального исследования на более широком наборе реальных данных. Примером таких данных являются графы MemoryAliases [21]
- Интеграция с одной из графовых баз данных. Примером такой базы является графовая база данных Neo4j.

Реализация представлена в репозитории:
https://github.com/vadyushkins/kotgll.

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. — 10 p. — URL: 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. — URL: https://doi.org/10.1145/2463664.2465216.

[3] Barrett Chris, Jacob Riko, Marathe Madhav. Formal-Language-Constrained Path Problems // SIAM Journal on Computing. — 2000. — Vol. 30, no. 3. — P. 809–837. — https://doi.org/10.1137/S0097539798337716.

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

[5] Context-Free Path Querying with Single-Path Semantics by Matrix Multiplication / Arseniy Terekhov, Artyom Khoroshev, Rustam Azimov, Semyon Grigorev // Proceedings of the 3rd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). — GRADES-NDA'20. — New York, NY, USA : Association for Computing Machinery, 2020. — 12 p. — URL: https://doi.org/10.1145/3398682.3399163.

[6] 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. — URL: https://doi.org/10.1145/3335783.3335791.

[7] 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. — URL: https://doi.org/10.1145/2499370.2462159.

[8] 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. — 7 p. — URL: https://doi.org/10.1145/3166094.3166104.

[9] 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. — ACM, 2017. — oct. — URL:

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

[11] Hellings Jelle. Explaining Results of Path Queries on Graphs // Software Foundations for Data Interoperability and Large Scale Graph Data Analytics / Ed. by Lu Qin, Wenjie Zhang, Ying Zhang et al. — Cham : Springer International Publishing, 2020. — P. 84–98.

[12] 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. — URL: https://doi.org/10.1145/3167132.3167265.

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

[14] Multiple-Source Context-Free Path Querying in Terms of Linear Algebra / Arseniy Terekhov, Vlada Pogozhelskaya, Vadim Abzalov et al. // International Conference on Extending Database Technology. — 2021.

[15] Shemetova Ekaterina, Azimov Rustam, Orachev Egor et al. One Algorithm to Evaluate Them All: Unified Linear Algebra Based Approach to Evaluate Both Regular and Context-Free Path Queries. — 2021. — P. 23.

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

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

[18] Santos Fred, Costa Umberto, Musicante Martin. A Bottom-Up Algorithm for Answering Context-Free Path Queries in Graph Databases. — 2018. — 01. — P. 225–233. — ISBN: 978-3-319-91661-3.

[19] Scott Elizabeth, Johnstone Adrian. GLL parsing // Electr. Notes Theor. Comput. Sci. — 2010. — 09. — Vol. 253. — P. 177–189.

[20] Tomita Masaru. LR Parsers for Natural Languages // Proceedings of the 10th International Conference on Computational Linguistics and 22nd Annual Meeting on Association for Computational Linguistics. — ACL '84/COLING '84. — USA : Association for Computational Linguistics, 1984. — P. 354–357. — URL: https://doi.org/10.3115/980491.980564.

[21] Zheng Xin, Rugina Radu. Demand-Driven Alias Analysis for C // Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '08. — New York, NY, USA : Association for Computing Machinery, 2008. — P. 197–208. — URL: https://doi.org/10.1145/1328438.1328464.

[22] Выпускная квалификационная работа “Реализация и экспериментальное исследование алгоритма поиска путей с контекстно-свободными ограничениями в графовой базе данных Neo4j“ Погожельской В.В. — URL: https://oops.math.spbu.ru/SE/diploma/2022/pi/Pogozhelskaya-report.pdf. (accessed: 05.04.2023).