Санкт-Петербургский государственный университет

ПОРСЕВ Денис Витальевич

Выпускная квалификационная работа

Алгоритм в терминах линейной алгебры для поиска путей от нескольких стартовых вершин с регулярными ограничениями

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5006.2019 «Математическое обеспечение и администрирование информационных систем»*

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

Рецензент:
Эксперт ООО «Техкомпания Хуавэй» Моисеев С. В.

Санкт-Петербург
2023

Saint Petersburg State University

Denis Porsev

Bachelor's Thesis

Linear algebra based multiple-source regular path querying algorithm

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2019 "Software and Administration of Information Systems"*

Scientific supervisor:
C.Sc., docent Semyon Grigorev

Reviewer:
Expert at Huawei Technologies Co. Ltd. Stanislav Moiseev

Saint Petersburg
2023

Оглавление

Введение

Современные компьютерные архитектуры позволяют легко обрабатывать линейные и иерархические структуры данных, такие как списки, стеки и деревья. Однако часто модель хранения данных представлена в виде графа. Такое представление применяется во множестве сфер: анализе социальных сетей, семантических сетей, анализе потока управления программ, биоинформатике и многих других.

Задача обработки данных, которые представлены в виде графа, имеет неструктурированный характер. Несмотря на это, некоторые алгоритмы анализа графов можно преобразовать к последовательности матрично–векторных операций, представив граф в виде матрицы смежности. Это позволяет выразить алгоритмы в терминах линейной алгебры, благодаря чему на практике за счет лучшего распараллеливания задачи удается получить эффективные реализации этих алгоритмов.

Более того, матрица смежности может быть представлена в разреженном виде, что позволяет сократить объем используемой памяти, так как в матрице хранятся только ненулевые элементы. Библиотеки разреженной линейной алгебры позволяют удобно работать с таким представлением. Они предоставляют набор примитивных операций таких, как матричное умножение, сложение, транспонирование, умножение вектора на матрицу и других, для работы с данными в разреженном виде.

Большую популярность в этом направлении набирает `GraphBLAS` [3] — стандарт, описывающий набор примитивных операций над матрицами и векторами, которые необходимы для выражения алгоритмов анализа графов в терминах линейной алгебры [15]. Существует несколько реализаций этого стандарта, как, например, `SuiteSparse:GraphBLAS` [2], которые удобно использовать для создания высокоэффективных алгоритмов анализа графов.

Одни из таких алгоритмов — алгоритмы поиска путей в графе. В частности, в терминах линейной алгебры могут быть выражены алгоритмы поиска кратчайшего пути в графе, поиска путей между мно-

жеством начальных и конечных вершин. В случае, когда важно знать только о существовании пути между вершинами, говорят, что решают задачу достижимости в графе. При этом на множество искомых путей в графе могут накладываться дополнительные ограничения с помощью формальных языков. Для этого требуют, чтобы метки на ребрах искомого пути при конкатенации образовывали слова, принадлежащие заданному над некоторым алфавитом формальному языку.

Для создания ограничений используют регулярные и контекстно–свободные языки. Регулярные ограничения на пути в графе были впервые введены в работах [6, 9] и нашли широкое применение в языках запросов к графовым базам данных [10], которые позволяют эффективно хранить и обрабатывать связанные данные, такие как социальные сети, сети транспорта и многое другое. Контекстно–свободные языки позволяют выразить более широкий класс ограничений и используются также в областях статического анализа кода [4], биоинформатике [13] и других.

Запросы с регулярными ограничениями (RPQ) являются хорошо изученными. Тем не менее в работе [16] было исследовано два подхода к решению задачи RPQ в языке запросов к графовым базам данных `SPARQL` [12]. Авторы работы выявили, что каждый из этих подходов эффективен только для конкретных видов графов.

В рамках учебной практики третьего курса автором данной работы был предложен новый алгоритм для решения задачи RPQ от множества стартовых вершин. Он был разработан на основе алгоритма поиска в ширину и реализован через примитивные операции разреженной линейной алгебры с помощью GraphBLAS. Для полученного алгоритма требуется провести экспериментальное исследование и сравнить его с аналогичными решениями.

1. Постановка задачи

Целью работы является проведение экспериментального исследования алгоритма для решения задачи RPQ, который нужно сравнить с аналогичными системами.

Для выполнения этой цели перед автором были поставлены следующие задачи.

- Выполнить обзор и выбрать множество инструментов, решающих задачу достижимости с регулярными ограничениями, для проведения сравнения с ними.
- Подготовить набор данных, состоящий из графов и регулярных запросов, для проведения экспериментального исследования.
- Спроектировать инструмент для автоматизации экспериментов.
- Провести экспериментальное исследование и проанализировать его результаты.

2. Обзор предметной области

В данном разделе приводится терминология используемая в работе, представляются основные подходы для решения задачи поиска путей с регулярными ограничениями.

2.1. Основы теории формальных языков

В этой секции вводятся ключевые понятия из теории формальных языков, которые нужны для введения ограничений на пути в графе с помощью регулярных языков.

Определение 2.1. *Формальной грамматикой* называется четверка $\langle V_N, V_T, P, S \rangle$, где

- V_N, V_T — конечные и непересекающиеся алфавиты нетерминалов и терминалов соответственно;
- P — конечное множество правил;
- S — стартовый нетерминал.

Определение 2.2. *Праволинейной грамматикой* называется формальная грамматика, правила которой могут быть заданы как $A \to aB$, $A \to a$, $A \to \epsilon$.

Определение 2.3. *Детерминированным конечным автоматом без эпсилон-переходов* называется пятерка $\langle Q, \Sigma, P, Q_s, F \rangle$, где

- Q — конечное непустое множество состояний;
- Σ — конечный входной алфавит;
- P — отображение $Q \times \Sigma \to Q$;
- $Q_s \subseteq Q$ — множество начальных состояний;
- $F \subseteq Q$ — множество конечных состояний.

Далее, будем называть детерминированный конечный автомат без эпсилон-переходов — конечным автоматом.

Определение 2.4. *Регулярным выражением* над алфавитом Σ называется

- a, если a — пустая строка, или $a \in \Sigma$;
- $e \cdot f$ (конкатенация), если e и f — регулярные выражения;
- $e \mid f$ (перечисление), если e и f — регулярные выражения;
- e^* (звезда Клини), если e — регулярное выражение.

Для любого регулярного выражения существует представление в виде конечного автомата. Этот факт используется при построении алгоритмов, основанных на выражении регулярных ограничений с помощью конечных автоматов.

2.2. Основные термины из теории графов

Определение 2.5. *Матрицей смежности* графа $\mathcal{G}$ называется матрица $M^{n \times n}$, где n — число вершин в графе, ячейка $M[i, j]$ имеет значение l, если существует ребро e между вершинами i, j с меткой l.

Заметим, что по каждой метке l графа $\mathcal{G}$ можно построить матрицу смежности $\mathcal{M}^l$. Тогда ячейки матрицы $\mathcal{M}^l$ будут отражать факт наличия ребра e с конкретной меткой l. Такое представление называется булевой декомпозицией матриц смежности графа. Оно позволяет хранить граф в виде набора булевых матриц, которыми удобно оперировать при построении алгоритмов на основе операций линейной алгебры.

Теперь можно сформулировать задачу поиска путей в графе с ограничениями, которые заданы с помощью регулярных языков. В частности, в данной работе рассматривается один из типов задачи поиска путей — задача достижимости. Сформулируем эту задачу двумя способами. Они отличаются в постановке результата, который ожидается в ходе решения задачи достижимости.

Определение 2.6. *Задача достижимости с ограничениями в виде регулярных языков.* Пусть имеются:

- граф $\mathcal{G} = \langle V, E, L \rangle$;
- регулярный язык $\mathcal{L}$;
- множество стартовых $V_s \subseteq V$ и финальных $V_f \subseteq V$ вершин.

Рассмотрим пути $\pi = (v_0, e_0, \ldots, e_n, v_n), e_k = (v_{k-1}, l_k, v_k)$. Сопоставим каждому пути слово $W(\pi) = (l_0 l_1 \ldots l_{n-1}) \in \mathcal{L}$. Необходимо:

- Найти *множество*, состоящее из вершин $v_n \in V$, для которых существует хотя бы один путь π с началом в $v_0 \in V$ такой, что $W(\pi) \in \mathcal{L}$, $v_0 \in V_s$, $v_n \in V_f$.
- Найти все *пары* вершин $v_0, v_n \in V$, для которых существует хотя бы один путь π такой, что $W(\pi) \in \mathcal{L}$, $v_0 \in V_s$, $v_n \in V_f$.

2.3. Существующие решения

В литературе выделяют несколько основных подходов к решению задачи поиска путей с ограничениями в виде регулярных языков. Они включают в себя подходы на основе конечных автоматов, использование программ на языке Datalog, построение индекса путей в графе для последующего исполнения регулярного запроса на основе индекса. Каждый из этих подходов будет подробнее рассмотрен в этой части обзора.

Алгоритмы для решения задачи PRQ на основе конечных автоматов строятся следующим образом. Входные данные в виде графа и регулярного языка представляются с помощью конечных автоматов. Для начала строится конечный автомат, который будет задавать регулярный язык. Далее, входной граф представляется в виде конечного автомата, у которого все состояния являются стартовыми и финальными. В итоге задачу поиска путей в графе с ограничениями, заданными с помощью

регулярных языков, можно свести к задаче нахождения пересечения двух конечных автоматов.

Также вычислить пересечение конечных автоматов можно с использованием произведения Кронекера, которое применяется к матрицам из булевых декомпозиций матриц смежности для входного графа и представления конечного автомата в виде другого графа. Подробнее, использование матричных операций и произведения Кронекера рассматривается в разделе 2.4.

2.3.1. Подход с использованием языка Datalog

Программа на Datalog представляет собой конечное множество правил. Правило — это выражение следующей формы.

$$h :- p_1, p_2, \ldots, p_n.$$

Выражения $h, p_1, \ldots, p_n$ представляют собой формулы вида $R(t_1, \ldots, t_k)$, где t_i может быть либо константой, либо переменной. Выражение p_i — факт, если все его t_i являются константами.

Для того, чтобы исполнять регулярные запросы с помощью программ на языке логического программирования Datalog, нужно представить граф в виде списка фактов о его ребрах.

$$edge(u_1, l_1, v_1)$$
$$edge(u_2, l_2, v_2)$$
$$\ldots$$
$$edge(u_n, l_n, v_n)$$

Где каждое ребро с меткой l_i, соединяет вершины u_i v_i.

Пример запроса l^*, который вычисляет транзитивное замыкание графа, содержащее рёбра с меткой l можно описать следующим образом.

$$path(x, y) :- edge(x, l, z), path(z, y).$$
$$path(x, y) :- edge(x, l, y).$$

Имея представление запроса в виде регулярного выражения, можно получить программу на языке Datalog, которая будет исполнять этот запрос. Для этого нужно представить регулярное выражение с помощью правил грамматики, после чего транслировать грамматику во множество правил Datalog.

Так, регулярное выражение l^* может быть представлено правилами $S \to l\ S$ и $S \to \epsilon$, которые легко транслируются в представленные выше правила Datalog.

Таким образом, для выполнения сравнения с этим подходом необходимо реализовать трансляцию регулярных выражений в программы на Datalog.

2.3.2. Подход, основанный на построении индекса

Существует множество работ на тему построения индекса путей в графе. Одни из них предлагают запоминать ключевые структуры в графе [17], как, например, часто встречающиеся подграфы или другие фрагменты, описывающие основные свойства графа. Другой подход основан на индексировании путей и используется в работе [5], где авторы хранят пути размера не больше k, k — целочисленная константа небольшого значения. Такой индекс позволяет исполнить любой регулярный запрос, состоящий не более чем из k меток, за один просмотр. Для запросов большего размера пути разделяются на части, длина которых кратна k, после чего запрос исполняется за несколько просмотров индекса.

Основным преимуществом построения индекса является то, что он позволяет быстро исполнить запрос по уже построенному пути. Однако использование индекса сказывается на размере потребляемой памяти во время работы алгоритма. Так, в исследовании [5] самый большой граф имеет 131828 вершин, и для него авторы не смогли построить индекс, длина пути которого больше 2, в виду ограничений по памяти.

2.4. Поиск путей с ограничениями с помощью операций над матрицами

Алгоритмы на основе матричных операций представляют еще один класс решений описанной задачи. Существует ряд алгоритмов, которые решают более общую задачу — задачу КС-достижимости, однако они могут быть применены к регулярным ограничения на пути в том числе.

2.4.1. Тензорный алгоритм

Один из таких алгоритмов основан на произведении Кронекера [1]. Он использует операции матричного умножения и не требует модификации изначальной контекстно-свободной грамматики.

На вход алгоритму подается конечный автомат, описывающий граф. Вторым аргументом алгоритм получает рекурсивный автомат, описывающий ограничения на пути в графе. Аналогично тому, как регулярное выражение может быть записано в виде конечного автомата, КС-грамматика может быть выражена с помощью рекурсивного автомата. Рекурсивный автомат может быть представлен в виде графа. Тогда, благодаря тензорному произведению, можно вычислить пересечение конечного автомата, соответствующего входному графу и рекурсивного автомата, соответствующего КС-ограничениям. После чего полученный автомат пересечения транзитивно замыкается. Эти операции применяются в цикле для булевой декомпозиции матриц смежности автомата, соответствующего КС-ограничениям, пока любая из матриц булевой декомпозиции меняется.

Операции вычисления транзитивного замыкания и тензорного произведения находят пути в графе для всех пар вершин. В случае, когда имеется множество стартовых вершин, производительность этого алгоритма может упасть, так как все вершины графа буду считаться начальными, и для всех них посчитаются пути до других вершин.

3. Алгоритм, основанный на операциях линейной алгебры и поиске в ширину

В рамках учебной практики третьего курса автором был предложен новый алгоритм для решения задачи достижимости, основанный на классическом алгоритме поиска в ширину для нескольких стартовых вершин [8].

Идея алгоритма состоит в последовательном применении операции умножения матриц для совершения шага в алгоритме поиска в ширину. На примере 3 показано, как шаг в алгоритме поиска в ширину выражается с помощью операции умножения матрицы на вектор.

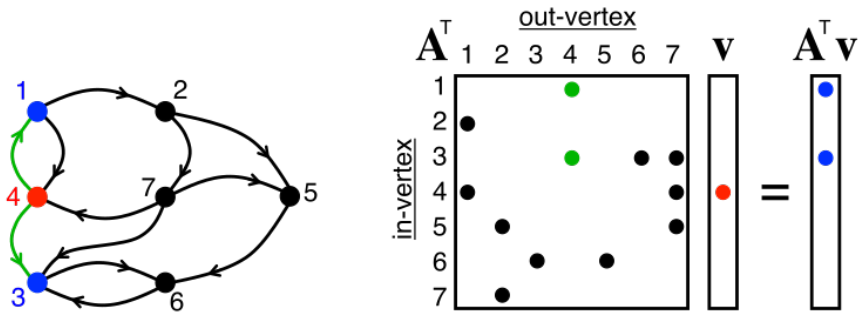


Рис. 1: Пример вычисления шага в алгоритме поиска в ширину, источник [7].

Представленный далее алгоритм расширяет эту идею и применяет её для одновременного обхода графа и автомата, выражающего ограничения, заданные с помощью регулярного языка.

3.1. Описание алгоритма

Пусть дан граф $\mathcal{G} = \langle V, E, L \rangle$, регулярный язык, описывающий ограничения на пути в нем, и множество начальных вершин V_s графа. Известно, что регулярный язык может быть представлен с помощью детерминированного конечного автомата. Также известно, что конечный автомат может быть представлен с помощью графа с дополнительной информацией о стартовых и финальных состояниях. Представим полученный граф в виде матрицы смежности. Тогда регулярные огра-

ничения могут быть выражены с помощью матрицы, ячейки которой описывают наличие перехода между двумя состояниями. Более того, для задачи достижимости достаточно хранить в матрице смежности на (i, j) месте два значения: 0 — отсутствие пути между вершинами i и j, 1 — наличие пути.

Пусть каждой метке l графа будет сопоставлена булева матрица смежности. Далее, будем оперировать с булевой декомпозицией матриц смежности $\mathcal{M}_A$ конечного автомата, и булевой декомпозицией матриц смежности $\mathcal{M}_G$ входного графа.

Алгоритм строит прямую сумму для матриц смежности $\mathcal{G}$ и $\mathcal{R}$. Представление в виде прямой суммы позволяет одновременно совершать шаг в алгоритме поиска в ширину на графе $\mathcal{G}$ и проверять удовлетворение регулярным ограничениям. Для каждого символа из пересечения множеств меток графа и алфавита регулярного языка построим матрицу $\mathcal{D} = \mathcal{M}_A \bigoplus \mathcal{M}_G$.

$$\mathcal{D} = \begin{bmatrix} \mathcal{M}_A & 0 \\ 0 & \mathcal{M}_G \end{bmatrix} \tag{1}$$

Далее, опишем процедуру обхода, основанную на серии умножений матрицы смежности. В классической версии обхода используется вектор v, куда записывается фронт обхода графа. Так, один раз перемножая матрицу смежности на вектор v, можно совершать один шаг в обходе графа, как было показано в примере 3.

Пусть k — количество вершин в графе, построенном по конечному автомату, n — количество вершин в графе $\mathcal{G}$. Тогда первые k элементов вектора v хранят информацию о номере состояния конечного автомата, построенного по $\mathcal{R}$, оставшиеся n элементов хранят информацию о номере вершины входного графа. При этом для проверки путей на соответствие входным регулярным ограничениям необходимо иметь информацию о парах (v_i, v_j), $0 <= i < k, k <= j < k + n$. Каждая из этих пар хранит информацию о том, в каком состоянии v_i мы оказались, достигнув вершину v_j. Предлагается разложить вектор v во множество векторов $v^0, v^1, ...v^k$ и получить матрицу M вида.

$$M^{k\times(k+n)} = \begin{bmatrix} v^0 \\ v^1 \\ \dots \\ v^k \end{bmatrix} \tag{2}$$

Построенная таким образом матрица M содержит информацию о достигнутых во время обхода вершинах графа. Остается лишь ограничить левую часть векторов v^i таким образом, чтобы они всегда содержали только одну единицу, что позволит определить в каком состоянии вершина была достигнута. Для этого скажем, что $v^i[i] = 1$, $v^i[j] = 0$ при $i \neq j$. Тогда v^i будет соответствовать i номеру состояния $\mathcal{R}$, и правая часть вектора v^i будет содержать информацию о достижимых вершинах. Матрица M примет вид.

$$M^{k\times(k+n)} = \begin{bmatrix} Id_k & Matrix_{k\times n} \end{bmatrix} \tag{3}$$

Где Id_k — единичная матрица, $Matrix_{k\times n}$ — матрица, хранящая фронт вершин в графе для каждого состояния.

В итоге остается проинициализировать множество начальных вершин. Для этого нужно, чтобы в ячейке $(i, i+j+1)$ матрицы M стояла 1, если состояние конечного автомата, построенного по $\mathcal{R}$, соответствующее номеру i, является начальным, а вершина графа $\mathcal{G}$ соответствующая номеру j содержится в множестве начальных вершин, то есть $j \in V_s$.

Перейдем к построению алгоритма, который представлен на листинге 1. На вход алгоритм принимает граф и ограничения в виде регулярного языка, которые можно представить с помощью конечного автомата.

Алгоритм обхода заключается в последовательном умножении матрицы M текущего фронта на матрицу $\mathcal{D}$. В результате чего, получается матрица M' содержащая информацию о вершинах, достижимых на следующем шаге. Далее, с помощью операций перестановки и сложения векторов M' преобразуется к виду матрицы M и присваивается ей для совершения нового шага. Важно отметить, что на каждом шаге хра-

Algorithm 1 Алгоритм в терминах линейной алгебры для поиска путей от нескольких стартовых вершин с регулярными ограничениями

1: **procedure** MSBFS ($\mathcal{R} = \langle Q, \Sigma, P, F, Q_s \rangle, \mathcal{G} = \langle V, E, L \rangle, V_s$)
2: $k \leftarrow |Q|$, $n \leftarrow |V|$
3: $\mathcal{M}_A \leftarrow$ булева декомпозиция матрицы смежности для $\mathcal{R}$
4: $\mathcal{M}_G \leftarrow$ булева декомпозиция матрицы смежности для $\mathcal{G}$
5: **for all** $q \in Q_s$ **do**
6: **for all** $v \in V_s$ **do**
7: $M[q, q + v + 1] \leftarrow 1$ ▷ Где $M^{k \times (k+n)}$ с 1 на главной диагонали
8: **for all** $a \in (\Sigma \cap L)$ **do**
9: $\mathcal{D}_a \leftarrow \mathcal{M}_A \bigoplus \mathcal{M}_G$
10: $M' \leftarrow M$, $M_{all} \leftarrow M$
11: **while** Матрица M_{all} меняется **do**
12: $M \leftarrow M' \langle \neg M_{all} \rangle$
13: **for all** $a \in (\Sigma \cap L)$ **do**
14: $M' \leftarrow M$ any.pair $\mathcal{D}_a$ ▷ Матр. умножение в полукольце
15: $M' \leftarrow TransformRows(M')$ ▷ Приведение M' к виду M
16: $M_{all} \leftarrow M'$
17: **return** M_{all}

нится матрица M_{all}, которая наполняется значениями, полученными на текущем шаге. Итерации продолжаются пока меняется M_{all}.

В алгоритме 1 на 15 строке происходит трансформация строчек в матрице M'. Это делается для того, чтобы представить полученную во время обхода матрицу M', содержащую новый фронт, в виде матрицы M для совершения последующих шагов. Для этого требуется так переставить строчки M', чтобы она содержала корректные по определению M значения. То есть, имела единицы на главной диагонали, а все остальные значения в первых k столбцах были нулями. Подробнее эта процедура описана в листинге 2.

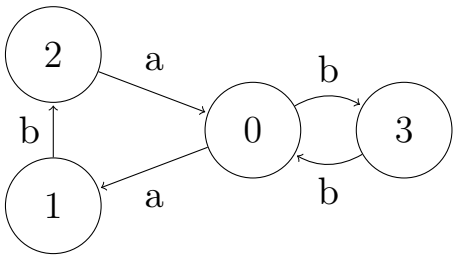
Algorithm 2 Алгоритм трансформации строчек

```
1: procedure TransformRows(M)
2:     T ← extractLeftSubMatrix(M)
3:     Ix, Iy ← итераторы по индексам ненулевых элементов T
4:     for i ∈ 0 . . . |Iy| do
5:         R ← M.getRow(Ix[i])
6:         M'.setRow(Iy[i], R + M'.getRow(Iy[i]))
```

В итоге алгоритм 1 решает задачу достижимости с регулярными ограничениями, так как матрица M_{all} содержит всю необходимую информацию о достигнутых вершинах. Для всех вершин f, соответствующих конечным состояниям F автомата, построенного по $\mathcal{R}$, в ячейке $M_{all}[f, f + i + 1]$ будет храниться единица, если вершина i во входном графе достижима из множества начальных вершин. Результатом работы алгоритма является матрица M_{all}, которая хранит информацию о множестве достигнутых вершин.

3.2. Пример

Рассмотрим работу алгоритма, представленного на листинге 1, на примере. Пусть имеется граф.

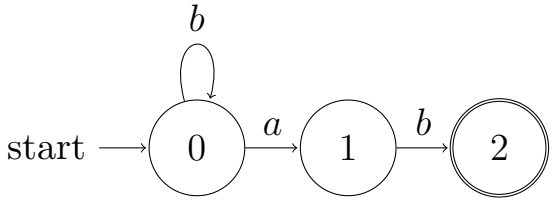


Для него алгоритм инициализирует следующие матрицы: матрицу смежности M_G и её булеву декомпозицию по каждой метке.

$$M_G = \begin{pmatrix} . & \{a\} & . & \{b\} \\ . & . & \{b\} & . \\ \{a\} & . & . & . \\ \{b\} & . & . & . \end{pmatrix}$$

$$\mathcal{M}_{G\_a} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad \mathcal{M}_{G\_b} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Зададим ограничения с помощью регулярного выражения b^*ab, которое представляется автоматом из трех последовательных состояний.



Автомат может быть задан матрицей смежности (с дополнительной информацией о стартовых и финальных состояниях). Для регулярного выражения b^*ab булева декомпозиция матрицы смежности выглядит следующим образом (при этом нужно запомнить, что состояние 0 является начальным, 2 — конечным).

$$M_A = \begin{pmatrix} \{b\} & \{a\} & . \\ . & . & \{b\} \\ . & . & . \end{pmatrix}$$

$$\mathcal{M}_{A\_a} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \mathcal{M}_{A\_b} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Для синхронизации обхода алгоритм в строчках 8–9 составляет набор блочно-диагональных матриц, каждая из которых — это прямая сумма двух матриц булевой декомпозиции: $\mathcal{D}_a = \mathcal{M}_{A\_a} \oplus \mathcal{M}_{G\_a}$ и $\mathcal{D}_b = \mathcal{M}_{A\_b} \oplus \mathcal{M}_{G\_b}$.

$$\mathcal{D}_a = \overbrace{\left(\begin{array}{ccc:cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right)}^{automaton} \underbrace{\phantom{\left(\begin{array}{cccc} 0 & 0 & 0 & 0 \end{array}\right)}}_{graph}$$

$$\mathcal{D}_b = \overbrace{\left(\begin{array}{ccc:cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right)}^{automaton} \underbrace{\phantom{\left(\begin{array}{cccc} 0 & 0 & 0 & 0 \end{array}\right)}}_{graph}$$

Пусть мы решаем частный случай задачи достижимости с несколькими стартовыми вершинами (multiple-source) — достижимость с одной стартовой вершиной (single-source). Пусть единственной начальной вершиной в графе будет вершина 0.

В строчках 5–7 алгоритм инициализирует матрицу M. Для этого единицами заполняются ячейки на главной дианогали, а также те ячейки, которые соответствуют вершинам из множества стартовых.

$$M = \begin{array}{|ccc|cccc|} \hline 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

На первой итерации алгоритма происходит умножение матриц M и $\mathcal{D}$ для совершения одного шага в обходе графа.

$$a: \begin{array}{|ccc|cccc|} \hline 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ \hline \end{array} \times \left(\begin{array}{ccc:cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) = \begin{array}{|ccc|cccc|} \hline 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ \hline \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} \\ \hline \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} & \cancel{0} \\ \hline \end{array}$$

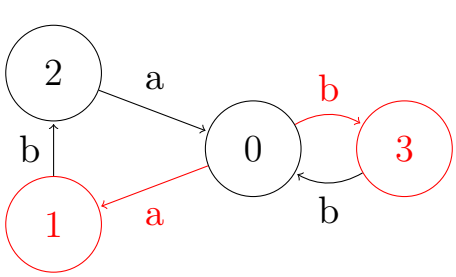
$$b: \begin{array}{|c|c|}\hline 1\,0\,0 & 1\,0\,0\,0 \\ \hline 0\,1\,0 & 0\,0\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,0\,0 \\ \hline \end{array} \times \left(\begin{array}{ccc:cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right) = \begin{array}{|c|c|}\hline 1\,0\,0 & 0\,0\,0\,1 \\ \hline 0\,0\,1 & 0\,0\,0\,0 \\ \hline \cancel{0\,0\,0} & \cancel{0\,0\,0\,0} \\ \hline \end{array}$$

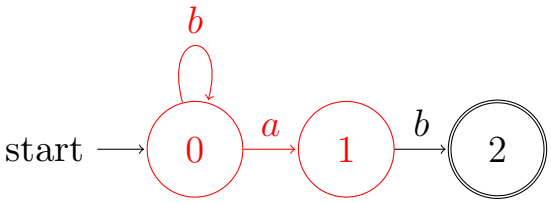
Для того, чтобы левая часть матрицы M всегда оставалась единичной, в строке 15 алгоритма происходит трансформация строчек матрицы M', которая в итоге присваивается M в строчке 12. Для этого происходит сложение тех строчек матрицы M', у которых в левой части единицы стоят на одинаковых позициях. После чего в матрице M' строчки переставляются так, чтобы левая часть M' принимала единичный вид. Строчки с пустой левой частью при этом не рассматриваются. После этих трансформаций правая часть матрицы M' кодирует фронт обхода графа для каждого состояния конечного автомата.

В нашем примере полученная матрица M' для следующего шага обхода выглядит следующим образом. В матрицу M_{all} при этом записывается накопленный результат.

$$M_{all} = M' = \begin{array}{|c|c|}\hline 1\,0\,0 & 0\,0\,0\,1 \\ \hline 0\,1\,0 & 0\,1\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,0\,0 \\ \hline \end{array}$$

Видно, что во фронт обхода графа попали вершины 1 и 3. В вершину 1 мы попали в состоянии 1, в вершину 3 — в состоянии 0. Совершаются следующие переходы в графе и автомате.



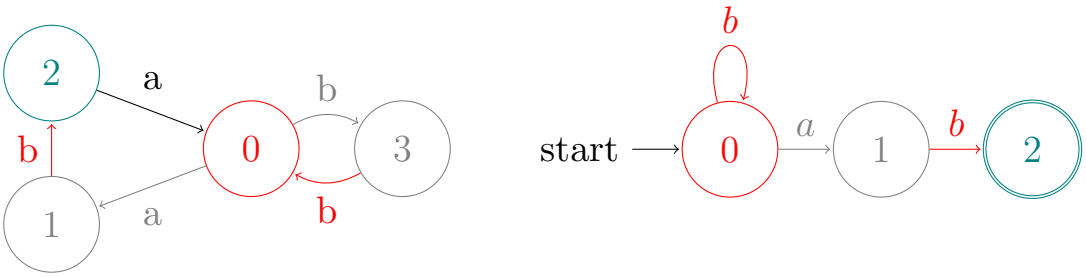


Совершим еще одну итерацию алгоритма.

$$a: \left(\begin{array}{c|c} 1\,0\,0 & 0\,0\,0\,1 \\ \hline 0\,1\,0 & 0\,1\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,0\,0 \end{array}\right) \times \left(\begin{array}{ccc:cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) = \left(\begin{array}{c|c} 0\,1\,0 & 0\,0\,0\,0 \\ \hline \cancel{0\,0\,0} & \cancel{0\,0\,0\,0} \\ \hline \cancel{0\,0\,0} & \cancel{0\,0\,0\,0} \end{array}\right)$$

$$b: \left(\begin{array}{c|c} 1\,0\,0 & 0\,0\,0\,1 \\ \hline 0\,1\,0 & 0\,1\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,0\,0 \end{array}\right) \times \left(\begin{array}{ccc:cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hdashline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right) = \left(\begin{array}{c|c} 1\,0\,0 & 1\,0\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,1\,0 \\ \hline \cancel{0\,0\,0} & \cancel{0\,0\,0\,0} \end{array}\right)$$

$$M' = \left(\begin{array}{c|c} 1\,0\,0 & 1\,0\,0\,0 \\ \hline 0\,1\,0 & 0\,0\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,1\,0 \end{array}\right) \qquad M_{all} = \left(\begin{array}{c|c} 1\,0\,0 & 1\,0\,0\,1 \\ \hline 0\,1\,0 & 0\,1\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,1\,0 \end{array}\right)$$



Видно, что мы достигли вершины 2 графа в конечном состоянии 2 автомата. При этом вершины графа 0, 1, 3 также достигнуты, однако это происходит в состояниях автомата 0, 1, 0 соответственно, которые не являются конечными.

Результатом работы алгоритма является матрица M_{all}, в которой содержится вся необходимая информация о достигнутых вершинах. В частности, для конечного состояния 2: $M_{all}[2, 5] = 1$ означает, что мы получили множество достижимых вершин, состоящее из одной вершины 2. После этой итерации матрица M_{all} перестает меняться и алгоритм останавливается.

Таким образом, в данной главе было показано как может быть построен алгоритм, основанный на поиске в ширину, для решения задачи достижимости с ограничениями, выраженными с помощью регулярных языков.

4. Архитектурные особенности инструмента для запуска экспериментов

В данной главе предлагается обсудить основные особенности инструмента, разработанного для запуска экспериментов. Его функциональность определена следующими требованиями:

- *Возможность запускать различные реализации алгоритмов решения задачи поиска путей с регулярными ограничениями.* Хотя инструмент в первую очередь спроектирован для запуска экспериментов с выбранными ранее аналогами, в области постоянно появляются новые реализации алгоритмов, которыми в будущем можно будет расширить исследование.

- *Поддержка расширения и изменения датасета, на котором проводится исследование.* Доступность инструментов для добавления новых графов или запросов в датасет упрощает пользование инструментом и позволяет пользователю запустить алгоритмы на своем наборе данных.

В соответствии с этим был разработан инструмент, позволяющий автоматизировать запуск алгоритмов, загрузку графов, генерацию запросов и стартовых вершин. Последовательность действий, реализуемая этим инструментом представлена на диаграмме 4.

Перед запуском экспериментов инструмент генерирует данные для соответствующего сценария исполнения. Для этого вызывается `Chunk generator`, который выбирает множество начальных вершин графа, на котором будет запущена каждая из реализаций. После чего генерируется множество запросов по определенным заранее шаблонам. Так, `Query generator` поддерживает генерацию запросов нескольких видов: в виде регулярного выражения, в виде правил регулярной грамматики и в виде программы на языке Datalog. Создание запросов происходит с помощью конфигурационных файлов, в которых хранятся имена меток на ребрах графа и количество ребер с соответствующей меткой. Благодаря чему удаётся создать запросы с наиболее популярными метками

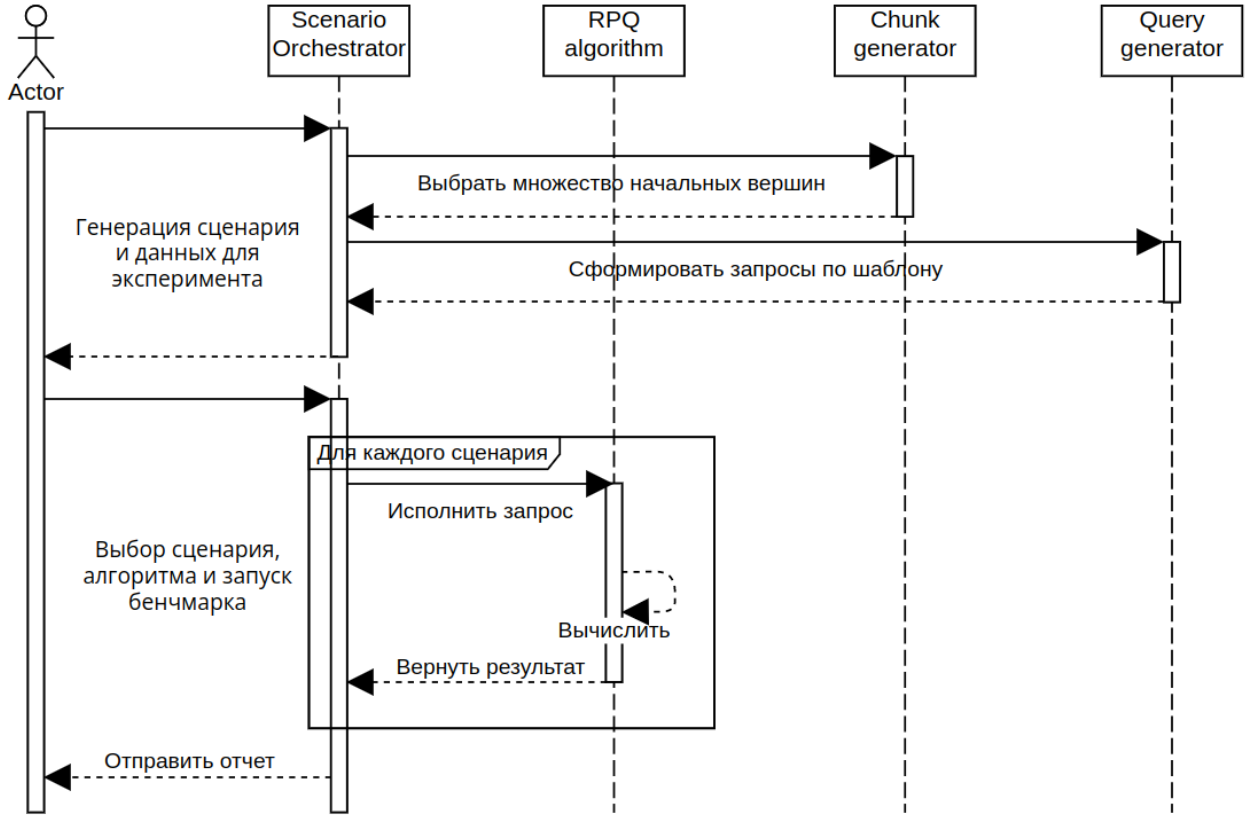


Рис. 2: Диаграмма последовательностей запуска экспериментов с помощью разработанного инструмента.

в графе. Шаблоны, по которым формируются запросы выделены в отдельный файл и могут быть изменены. Также, доступен инструмент для подсчета статистики меток в графе, который позволяет создать конфигурационный файл по каждому графу.

Далее, пользователь инициирует выполнение эксперимента на выбранном множестве сценариев. Для каждого из сценариев запускается реализация алгоритма, решающего задачу поиска путей с регулярными ограничениями. После того, как алгоритм закончил исполнение запроса, он возвращает результат в виде числа найденных вершин. В итоге `Scenario orchestrator` сохраняет показатели времени исполнения вычислений в отдельный файл, а также обрабатывает ошибки, которые могли возникнуть во время работы алгоритма. В результате постановки эксперимента отправляется отчет, содержащий информацию о запущенном сценарии, времени исполнения и результате работы алгоритма.

5. Эксперимент

Данный раздел посвящен описанию проводимого экспериментального исследования. В предыдущих главах были представлены основные подходы к исполнению запросов с регулярными ограничениями, а также описан новый алгоритм, основанный на операциях линейной алгебры и поиске в ширину. Сравнение производительности этого алгоритма с реализациями других подходов является основной целью данного исследования.

5.1. Исследовательские вопросы и метрики

В качестве основной метрики производительности рассматривается время работы алгоритмов на регулярных запросах. Цель экспериментального исследования — ответить на два исследовательских вопроса:

- **RQ1**: Какова производительность разработанного алгоритма по сравнению с существующими аналогами?
- **RQ2**: Как влияет размер множества стартовых вершин на производительность реализации разработанного алгоритма?

5.2. Окружение и конфигурации

Экспериментальное исследование проводится на сервере под управлением ОС Ubuntu 20.04 со следующими характеристиками: процессор Intel Core i7-6700 CPU, 4 ядра с частотой 3.40 ГГц; объем оперативной памяти 64 Гб.

Для проведения сравнения с аналогом на основе языка Datalog было принято решение использовать язык Souffle [14], который является одним из диалектов языка Datalog и преодолевает некоторые ограничения классического варианта. Souffle часто используется в задачах языкового анализа и имеет сейчас большую популярность. Установлена версия Souffle 2.4, которая была сконфигурирована с размером слова для примитивного типа 64 бита и запускалась с флагами

`-j {число ядер процессора}`, который используется для параллельного исполнения программы, и `--magic-transform=*` для преобразования программы в эквивалентную с потенциально меньшим числом промежуточных вычислений.

Для подхода, основанного на построении индекса, была найдена реализация[1], находящаяся в свободном доступе, которую автору удалось запустить, однако этот инструмент выдает неверный результат при запуске на сгенерированных по шаблонам запросах, по этой причине он был не включен в сравнительный анализ.

Реализация тензорного алгоритма для нескольких стартовых вершин была взята из репозитория CFPQ_PyAlgo[2].

5.3. Условия эксперимента

В качестве исходных данных были выбраны графы, собранные по реальным RDF данным и данным социальных сетей. Их характеристики представлены в таблице. В таблице выделено количество вершин и рёбер графа, а также, число уникальных меток, содержащихся на рёбрах. Графы класса RDF расположены в первой части таблицы, графы социальных сетей — во второй.

Graph	#V	#E	#L
enzyme	48 815	86 543	14
eclass	239 111	360 248	10
go	582 929	1 437 437	47
geospecies	450 609	2 201 532	158
taxonomy	5 728 398	14 922 125	21
advogato	6 541	51 127	3
youtube	15 088	27 257 790	5

Таблица 1: Графы.

Стоит отметить, что графы, собранные по данным социальных сетей являются достаточно плотными в сравнении с графами, построенными

[1]Repository for the prototype code for index RPQ paper: https://github.com/darroyue/Ring-RPQ (accessed: 20.05.2023)

[2]Repository for developing, testing and benchmarking CFPQ algorithms implemented in PyGraphBLAS: https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_PyAlgo (accessed: 20.05.2023)

по RDF-данным, так как имеют существенно большее число ребер, чем вершин. Это может привести к тому, что во время обхода этих графов будут пройдены более длинные пути, что может сказаться на времени исполнения алгоритмов.

Регулярные запросы к графам были сгенерированы на основе самых популярных шаблонов запросов, собранных в работе [11]. В запросы входят классические операторы регулярных выражений такие, как конкатенация и перечисление, а также звезда Клини и множество комбинаций этих операторов. Всего отобрано 16 шаблонов, они представлены в таблице 2.

Name	Query	Name	Query
q_0	a^*	q_8	$a \cdot b$
q_1	$a \cdot b^*$	q_9	$a \cdot b \cdot c$
q_2	$a \cdot b^* \cdot c^*$	q_{10}	$a \cdot b \cdot c \cdot d$
q_3	$a \cdot b^* \cdot c$	q_{11}	$(a \cdot b)^+ \mid (c \cdot d)^+$
q_4	$a^* \cdot b^*$	q_{12}	$(a \cdot (b \cdot c)^*)^+ \mid (d \cdot e)^+$
q_5	$a \cdot b \cdot c^*$	q_{13}	$(a \cdot b \cdot (c \cdot d)^*)^+ \mid (e \mid f)^*$
q_6	$(a \mid b \mid c \mid d \mid e)^+$	q_{14}	$(a \mid b)^+ (c \mid d)^+$
q_7	$(a \mid b \mid c \mid d \mid e) \cdot f^*$	q_{15}	$a \cdot b \cdot (c \mid d \mid e)$

Таблица 2: Шаблоны запросов.

Для каждого графа отбиралось 5 самых популярных меток, после чего по ним на основе шаблонов генерировалось случайным образом 5 запросов. Важно отметить, что в графе *advogato* меньше 5 различных меток, по этой причине в сгенерированных для этого графа запросах допускается повторение меток.

В зависимости от числа вершин, на которых запускаются регулярные запросы, их можно разделить на несколько видов:

- Single-source запросы, для которых алгоритм запускается от одной исходной вершины.
- Multiple-source запросы запускаются от определенного изначально множества вершин в графе.
- All-pairs запросы запускаются от множества всех вершин графа.

Для выбора множества начальных вершин было решено создавать случайные выборки из нужного числа вершин. Так, для single-source запросов создавалось множество из 100 начальных вершин, для каждой из которых запускался запрос. Для multiple-source запросов создавалась выборка из 2, 5, 10, 100, 1000, 10000 случайных вершин для каждого графа, на которой запускались запросы. При этом финальными считались все вершины графа. Скрипты для создания запросов и генерации вершин доступны в репозитории[3].

5.4. Результаты сравнения алгоритмов

RQ1: Какова производительность разработанного алгоритма по сравнению с существующими аналогами?

На рисунках 3–4 представлены результаты эксперимента по сравнению производительности алгоритмов на графах, построенных по RDF данным. Введены следующие обозначения: алгоритм, основанный на поиске в ширину, обозначен MSBFS, алгоритм, основанный на тензорном произведении — Tensor, решение, основанное на Datalog и реализованное с помощью диалекта Souffle — Datalog. Для всех шаблонов в виде гистограмм представлено среднее арифметическое значение времени исполнения алгоритмов с отклонением от среднего на соответствующем запросе.

Прежде всего эксперимент был поставлен на графах наименьшего размера — *enzyme* и *eclass* для случая single-source запросов, который представлен в первой колонке. Результаты показывают, что реализации алгоритмов в терминах матричных операций в среднем на порядок быстрее реализации на Datalog. В частности, алгоритм на основе тензорного произведения работает более чем в 10 раз быстрее аналога на Datalog, алгоритм MSBFS — в среднем в 100 раз быстрее. Важно отметить, что программы для аналога на Datalog генерировались по соответствующим представлениям регулярных ограничений в виде грамматик. Реализовать каждый из запросов для конкретного графа можно

[3]Benchmark suite for RPQ evaluation: https://github.com/bahbyega/paths-benchmark (accessed: 20.05.2023)

эффективнее, используя дополнительные средства языка Souffle, что могло повлиять на показатели времени, представленные на рисунках.

Далее, три реализаций запускались на multiple-source запросах. Результаты для наибольшего из множеств начальных вершин (10000 вершин) представлены на рисунках 3–4 во второй колонке. На множестве из 10000 стартовых вершин Tensor и Datalog показывают близкую производительность, проигрывая MSBFS. При этом показатели времени исполнения тензорным алгоритмом имеют более высокое отклонение от среднего значения. Это может объясняться тем, что конкретный шаблон представлен несколькими запросами с метками, расположенными в разном порядке. Из-за чего результат исполнения запроса может сильно отличаться по количеству достижимых вершин. Так, для шаблонов q_6, q_8, q_9, q_{10} большое значение имеет порядок, в котором расставлены метки в шаблоне, потому что в них отсутствует оператор звезды Клини.

После чего проводился эксперимент на графе *go*. Из него видно, что производительность MSBFS оказалась незначительно хуже алгоритма, основанного на тензорном произведении. Это может быть объяснено структурой графа *go*. Он имеет большое число ребер с одной и той же меткой, которая чаще всего встречалась в запросах, что повлияло на количество шагов алгоритма MSBFS.

По экспериментам на графах *geospecies* и *taxonomy* также можно выделить лучшую производительность MSBFS и Tensor относительно Datalog. Однако тензорный алгоритм демонстрирует выбросы на некоторых запросах, когда запрос запускается на множестве из 10000 вершин. Кроме того, важно отметить, что производительность алгоритмов зависит от сложности запроса. Так, запросы q_7 и q_{13}, содержащие в себе большее количество меток и операторов, чем остальные запросы, исполняются в среднем дольше остальных.

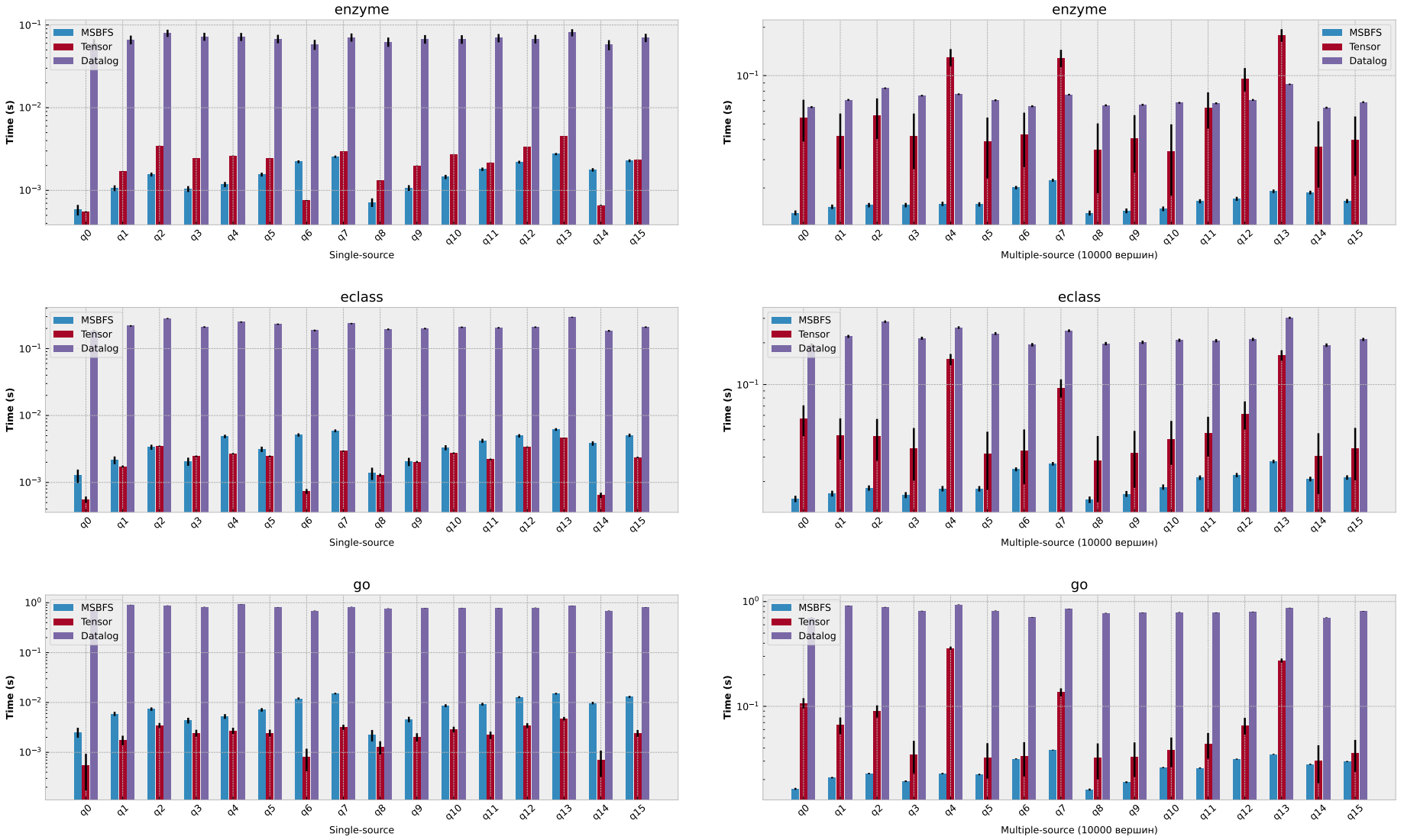


Рис. 3: Результаты эксперимента на наборе RDF-данных для single-source и multiple-source (10000 вершин) запросов.

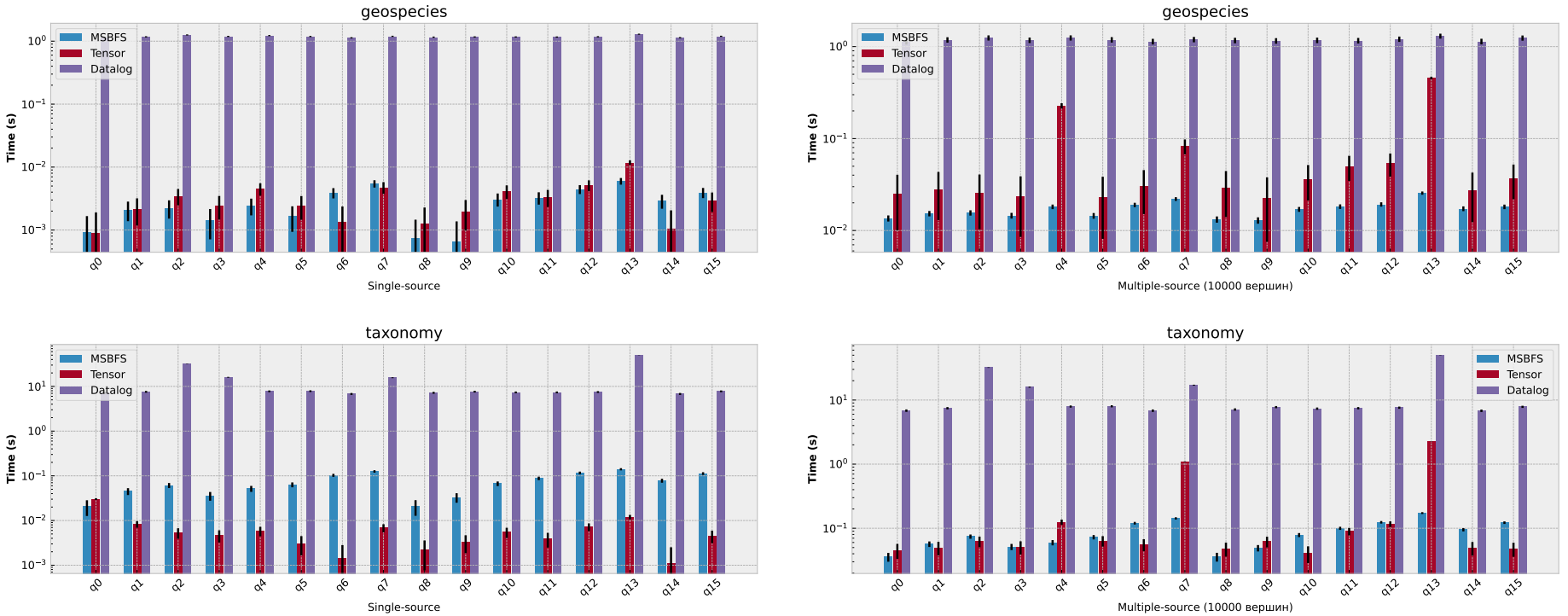


Рис. 4: Результаты эксперимента на наборе RDF-данных для single-source и multiple-source (10000 вершин) запросов.

RQ2: Как влияет размер множества стартовых вершин на производительность реализации разработанного алгоритма?

Результаты эксперимента по анализу производительности алгоритма для различного числа стартовых вершин представлены на рисунке 5.4. На нем изображена зависимость времени исполнения запроса от числа стартовых вершин в графе для алгоритмов MSBFS и Tensor. Для выявления зависимости числа стартовых вершин ко времени исполнения была взята еще одна реализация MSBFS, обозначенная MSBFSPairs, которая, в отличие от MSBFS, находит не множество достижимых вершин, а пары начальная–конечная вершина. Основной особенностью этой реализации является то, что инициализация матриц стартовыми вершинами увеличивает размер перемножаемых матриц в n раз, где n — число стартовых вершин. Это позволяет запомнить от какой стартовой вершины была достигнута каждая из вершин в конечном множестве достижимых. Отмечено время исполнения каждого из запросов, а также выделено среднее время их исполнения. Результаты показывают, что с ростом числа стартовых вершин время работы алгоритма MSBFS меняется несущественно. Это может объясняться тем, что инициализация матриц стартовыми вершинами в его реализации не влияет на размер перемножаемых матриц, что не создает дополнительных расходов во время работы алгоритма, в то время как алгоритмы Tensor и MSBFSPairs страдают от существенного снижения производительности при росте числа стартовых вершин.

Также, на основе этого рисунка можно сделать вывод о том, что время исполнения запроса зависит не только от размера графа, но и от его структуры и разреженности. Так, по анализу данных социальных сетей можно сделать вывод о том, что алгоритм на основе тензорного произведения демонстрирует лучшую производительность для более плотных графов, таких как *advogato* и *youtube*. Именно эти графы имеют наименьшее число различных меток, граф *youtube* также имеет наибольшее число ребер среди всех графов датасета при существенно меньшем числе вершин. Это приводит к тому, что алгоритму, основанному на поиске в ширину, приходится выполнять больше ша-

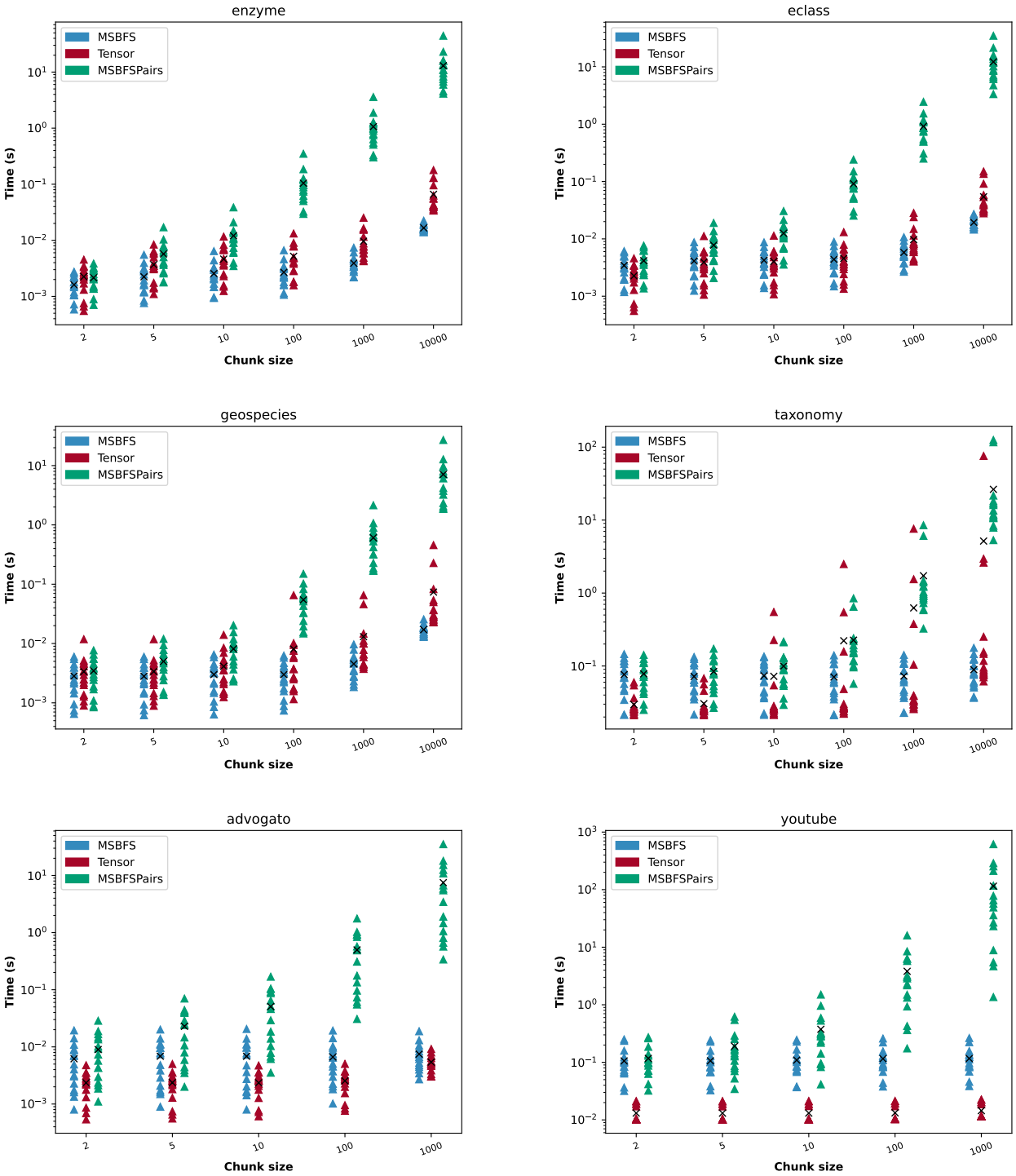


Рис. 5: Результаты эксперимента с multiple-source запросами для разных выборок стартовых вершин.

гов во время обхода графа, так как на этих графах с каждым шагом алгоритма более вероятно увеличение фронта обхода.

Результаты анализа экспериментального исследования позволяют заключить, что реализация алгоритма, основанного на поиске в ширину, показывает приемлемую производительность и во многих случаях работает быстрее, чем реализации аналогов. Более того, реализация алгоритма MSBFS для нахождения всего множества достижимых вершин показывает несущественное ухудшение производительности с ростом числа стартовых вершин. В то же время, реализация алгоритма MSBFSPairs, результатом которого является множество пар начальная–конечная вершина, показала худшие временные характеристики. Более того, заметным является падение производительности обоих реализаций MSBFS на графах социальных сетей, что может быть исследовано в дальнейшем на более широком наборе данных.

Заключение

При проведении данной работы были достигнуты следующие результаты.

- Проведен обзор и выбрано два доступных аналога для проведения сравнительного анализа, а именно, тензорный алгоритм и алгоритм на основе языка Datalog. Последний является принятым базовым инструментом.

- Собран набор данных, состоящий из графов и регулярных запросов, для проведения экспериментального исследования. В него вошли графы собранные по RDF-данным и данным социальных сетей, а также множество популярных запросов.

- Спроектирован инструмент, который позволяет автоматизировать проведение экспериментов, поддерживает генерацию данных для запуска запросов на графах, получение множества начальных вершин и запуск различных типов запросов[4].

- Проведено экспериментальное исследование алгоритма и сравнение с аналогами. В ходе анализа было выявлено, что подход к реализации алгоритма поиска путей с ограничениями в виде регулярных языков на основе поиска в ширину является жизнеспособным и показывает приемлемые результаты на реальных данных. Более того, на RDF-данных алгоритм показывает кратный рост производительности относительно аналога, реализованного на Datalog, а также работает в среднем быстрее алгоритма, основанного на тензорном произведении. Тем не менее, анализ зависимости времени исполнения запроса к числу стартовых вершин выявил проблемы реализации алгоритма MSBFS для решения задачи достижимости от каждой стартовой вершины, что может быть исследовано в дальнейшей работе.

[4]Benchmark suite for RPQ evaluation: https://github.com/bahbyega/paths-benchmark (accessed: 20.05.2023)

В дальнейшем работа может быть развита в следующих направлениях.

- Расширение набора данных, на котором исследуется алгоритм. В него можно включить искусственно сгенерированные графы с определенными особенностями структуры, а также реальные графы из других прикладных областей.

- Интегрирование алгоритма с графовой базой данных, а также с языком запросов к графовой базе данных. Это позволит провести еще более полноценное сравнение алгоритма с аналогами, в которые можно будет включить системы графовых баз данных.

Список литературы

[1] Context-Free Path Querying by Kronecker Product / Egor Orachev, Ilya Epelbaum, Rustam Azimov, Semyon Grigorev. — 2020. — 08. — P. 49–59.

[2] Davis Timothy A. Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra // ACM Trans. Math. Softw. — 2019. — dec. — Vol. 45, no. 4. — 25 p. — URL: https://doi.org/10.1145/3322125.

[3] Design of the GraphBLAS API for C / Aydin Buluç, Tim Mattson, Scott McMillan et al. // 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). — 2017. — P. 643–652.

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

[5] Fletcher George, Peters J., Poulovassilis Alexandra. Efficient regular path query evaluation using path indexes. — 2016. — 03.

[6] Florescu Daniela, Levy Alon, Mendelzon Alberto. Database Techniques for the World-Wide Web: A Survey // SIGMOD Rec. — 1998. — sep. — Vol. 27, no. 3. — P. 59–74. — URL: https://doi.org/10.1145/290593.290605.

[7] GraphBLAS. Wikipedia. The Free Encyclopedia. — https://en.wikipedia.org/wiki/GraphBLAS. — Accessed: 2023-05-11.

[8] A GraphBLAS solution to the SIGMOD 2014 Programming Contest using multi-source BFS / Márton Elekes, Attila Nagy, Dávid Sándor

et al. // 2020 IEEE High Performance Extreme Computing Conference (HPEC). — 2020. — P. 1–7.

[9] Mendelzon Alberto O., Wood Peter T. Finding Regular Simple Paths in Graph Databases // SIAM J. Comput. — 1995. — dec. — Vol. 24, no. 6. — P. 1235–1258. — URL: https://doi.org/10.1137/S009753979122370X.

[10] Nolé Maurizio, Sartiani Carlo. Regular Path Queries on Massive Graphs // Proceedings of the 28th International Conference on Scientific and Statistical Database Management. — SSDBM '16. — New York, NY, USA : Association for Computing Machinery, 2016. — 12 p. — URL: https://doi.org/10.1145/2949689.2949711.

[11] 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. — 03.

[12] Prud'hommeaux Eric, Seaborne Andy. SPARQL Query Language for RDF. — W3C Recommendation. — 2008. — URL: http://www.w3.org/TR/rdf-sparql-query/.

[13] Sevon Petteri, Eronen Lauri. Subgraph Queries by Context-free Grammars // Journal of Integrative Bioinformatics. — 2008. — Vol. 5, no. 2. — P. 157–172. — URL: https://doi.org/10.1515/jib-2008-100.

[14] Shahin Ramy, Chechik Marsha. Variability-Aware Datalog // Practical Aspects of Declarative Languages. — Springer International Publishing, 2020. — P. 213–221. — URL:

[15] Standards for graph algorithm primitives / Tim Mattson, David Bader, Jon Berry et al. // 2013 IEEE High Performance Extreme Computing Conference (HPEC). — 2013. — P. 1–2.

[16] Yakovets Nikolay, Godfrey Parke, Gryz Jarek. Query Planning for Evaluating SPARQL Property Paths // Proceedings of the 2016 International Conference on Management of Data. — SIGMOD '16. — New York, NY, USA : Association for Computing Machinery, 2016. — P. 1875–1889. — URL: https://doi.org/10.1145/2882903.2882944.

[17] Yan Xifeng, Han Jiawei. Graph Indexing // Managing and Mining Graph Data / Ed. by Charu C. Aggarwal, Haixun Wang. — Boston, MA : Springer US, 2010. — P. 161–180. — ISBN: 978-1-4419-6045-0. — URL: https://doi.org/10.1007/978-1-4419-6045-0_5.