Санкт-Петербургский государственный университет

***Илья Владимирович Эпельбаум***

Выпускная квалификационная работа

# Оптимизация алгоритма поиска путей с контекстно-свободными ограничениями, основанного на произведении Кронекера

Уровень образования: бакалавриат

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

Основная образовательная программа *СВ.5006.2017 «Математическое обеспечение и администрирование информационных систем»*

Профиль *Системное программирование*

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

Рецензент:
Программист, ООО «ИнтеллиДжей Лабс» Р.Ш. Азимов

Санкт-Петербург
2021

Saint Petersburg State University

***Ilya Epelbaum***

Bachelor's Thesis

# Optimization of the pathfinding algorithm with context-free constraints based on the Kronecker product

Education level: bachelor

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

Programme *CB.5006.2017 "Software and Administration of Information Systems"*

Profile: *Software Engineering*

Scientific supervisor:
C.Sc, docent. S.V. Grigoriev

Reviewer:
Software engineer, "JetBrains Research" R.S. Azimov

Saint Petersburg
2021

# Оглавление

# Введение

Одной из фундаментальных областей программной инженерии является разработка СУБД. На данный момент создано множество разных моделей построения таковых. Например, одной из них является графовая модель, в которой данные представлены в виде вершин — сущности и дуг — связи между сущностями. Примерами графовых СУБД являются RedisGraph[1], Neo4j[2] и TigerGraph[3].

К графовой базе данных необходимо выполнять запросы. В ходе их рассмотрения появляется задача поиска путей в графе с ограничениями, которые выразимы в терминах формальной грамматики. В таком случае каждому пути соответствует слово, полученное конкатенацией меток, находящихся на его дугах в порядке обхода. Именно это слово будет проверяться на принадлежность заданной грамматике.

В этой связи особый интерес представляют контекстно-свободные (КС) грамматики, так как обладают более выразительной способностью в отличии, например, от регулярных. То есть данный тип грамматики позволяет задавать более сложные ограничения на пути. Поэтому запросы с КС-ограничениями используют и в других областях, например, биоинформатика [11] или статический анализ кода [9].

Существует множество алгоритмов, решающих задачу поиска путей основываясь на теории формальных языков, например, алгоритм Элле Хеллингса [8], восходящий алгоритм Фреди Сантоса [10] или алгоритм Петтери Севона [11]. Однако, в статье Йохима Куйперса и др. [13] продемонстрировано, что существующие алгоритмы могут быть успешно применены лишь при определенных условиях на входные данные, а в общем случае их реализации требуют большой объем вычислительных мощностей в процессе работы, что не оправдывает их применение в промышленных продуктах. В то же время Никита Мишин и др. в статье [7] и Егор Орачев и др. в исследовании [5], взяв собранный набор

[1]Репозиторий проекта RedisGraph: https://github.com/RedisGraph/RedisGraph. Дата посещения: 20.04.2021.

[2]Сайт проекта Neo4j: https://neo4j.com. Дата посещения: 20.04.2021.

[3]Сайт проекта TigerGraph: https://www.tigergraph.com. Дата посещения: 20.04.2021.

данных *CFPQ_Data*[4], показали, что алгоритмы, основанные на линейной алгебре, а именно алгоритм, основанный на матричном умножении, и алгоритм, основанный на произведении Кронекера, демонстрируют высокую производительность на реальных входных данных.

Однако в исследовании [5] авторы замечают неоптимальность предложенного алгоритма, основанного на произведении Кронекера, что влечет потребность в улучшении текущего результата.

[4]Репозиторий набора данных *CFPQ_Data*, наполненный различными реальными и синтетическими графами: https://github.com/JetBrains-Research/CFPQ_Data. Дата посещения: 20.04.2021

# 1. Обзор

Рассмотрим алгоритмы, решающие задачу поиска путей с контекстно-свободными ограничениями с помощью операций линейной алгебры, в несколько этапов. Первым этапом приведем определения, встречающиеся в их описании. Вторым этапом рассмотрим алгоритм, основанный на произведении Кронекера, и алгоритм, основанный на матричном умножении, и сравним их, выделяя проблемы, которые лягут в основу работы. Последним этапом выберем инструмент для оценки производительности решений указанных проблем.

## 1.1. Базовые определения

Следующее определение формально описывает структуру представления грамматики для алгоритма, основанного на произведении Кронекера.

**Определение 1.1.** Рекурсивный автомат [1] над конечным алфавитом $\Sigma$ есть набор $(M, m, \{C_i\}_{i \in M})$, где

- $M$ конечное множество меток,
- $m \in M$ начальная метка,
- $\{C_i\}_{i \in M}$ множество *конечных автоматов*, где $C_i = (\Sigma \cup M, Q_i, q_i^0, F_i, \delta_i)$:

  $\Sigma \cup M$ множество символов, $\Sigma \cap M = \emptyset$,

  $Q_i$ конечное множество состояний, где $Q_i \cap Q_j = \emptyset, \forall i \neq j$,

  $q_i^0$ начальное состояние $C_i$,

  $F_i$ множество финальных состояний $C_i$, где $F_i \subseteq Q_i$,

  $\delta_i$ функция переходов $C_i$, где $\delta_i : Q_i \times (\Sigma \cup M) \to Q_i$.

**Определение 1.2.** Пусть $\mathcal{G}$ = (Σ,N,P,S) — контекстно-свободная грамматика, тогда $\mathcal{G}$ находится в **ослабленной нормальной форме Хомского** (ОНФХ), если содержит только правила вида:

- $A \to BC$, где $A$, $B$, $C \in N$
- $A \to a$, где $A \in N$, $a \in \Sigma$
- $A \to \varepsilon$, где $A \in N$

Определение ОНФХ отличается от нормальной формы Хомского наличием правил вида $A \to \varepsilon$, где A — любой нетерминал, то есть A не обязательно является стартовым, а также допущением использовать стартовый нетерминал в правых частях правил при наличии правила вида $S \to \varepsilon$.

**Определение 1.3.** Пусть $G$ — помеченный граф с n вершинами, $M$ — матрица смежности $G$ и $L$ — множество различных меток на дугах графа $G$, тогда матрицей смежности для $l \in L$ называется матрица $M^l$ размером $n \times n$, где

$$M^l[i,j] = \begin{cases} true, \text{ между вершинами i и j существует дуга с меткой l} \\ false, \text{ иначе} \end{cases}$$

## 1.2. Алгоритм, основанный на произведении Кронекера

В статье [5] был представлен алгоритм, основанный на произведении Кронекера, для поиска путей между всеми парами вершин, именуемый алгоритмом построения индекса, то есть результатом работы является структура данных, указывающая между какими парами есть требуемый путь. Представленный алгоритм основан на матричных операциях и использует рекурсивный автомат.

Алгоритм (листинг 2) принимает граф $G$ и рекурсивный автомат $R$. Основная идея заключается в использовании произведения Кронекера, как инструмента для пересечения двух автоматов. В качестве второго автомата в алгоритме предлагается выбрать граф $G$. Такой выбор является корректным, так как каждую вершину можно считать состоянием, а дуги — переходами.

На начальном этапе (строки 2-3) алгоритма берутся матрицы смежности для всех меток графа и автомата по определению 1.3. Стоит отметить, что определение 1.3 дано для графа, но аналогичное определение можно ввести и для рекурсивного автомата. Дополнительно (строки 4-5) для каждого нетерминала из $R$ создается пустая матрица смежности в наборе $M_2$, который соответствует графу $G$.

Первым шагом (строки 6-9) проверяется наличие состояний в автомате $R$, которые одновременно являются стартовыми и конечными, то есть проверяется наличие $\varepsilon$ правил для грамматики. При наличии таковых из каждой вершины в неё же есть искомый путь.

Основной цикл (строки 10-20) алгоритма выполняется до тех пор пока набор матриц смежности для графа меняется. Первым шагом цикла происходит вычисление произведения Кронекера (строка 11), тем самым создавая матрицу смежности нового автомата. Далее результат транзитивно замыкается для получения информации о достижимости состояний в полученном автомате. И последним шагом (строки 14-20) происходит обновление при помощи операций, представленных на листинге 1, матрицы графа, которая соответствует нетерминалу, конечное состояние автомата которого достижимо из начального состояния.

**Listing 1** Вспомогательные функции

```
1: function getStates(M_1, i, j)
2:     r ← dim(M_1)
3:     return ⌊i/r⌋, ⌊j/r⌋
4: function getCoordinates(M_2, i, j)
5:     n ← dim(M_2)
6:     return i mod n, j mod n
```

**Listing 2** Алгоритм, основанный на произведении Кронекера

```
1:  function contextFreePathQuerying(G, R)
2:      M_1 ← набор матриц смежности для R
3:      M_2 ← набор матрица смежности для G
4:      for all nonterminals ∈ R do
5:          M_2 = M_2 ∪ {пустая матрица}
6:      for s ∈ 0..dim(M_1) − 1 do
7:          for S ∈ getNonterminals(R, s, s) do
8:              for i ∈ 0..dim(M_2) − 1 do
9:                  M_2^S[i, i] ← {1}
10:     while Набор M_2 изменяется do
11:         M_3 ← M_1 ⊗ M_2
12:         C_3 ← transitiveClosure(M_3)
13:         n ← dim(M_3)
14:         for i ∈ 0..n − 1 do
15:             for j ∈ 0..n − 1 do
16:                 if C_3[i, j] then
17:                     s, f ← getStates(C_3, i, j)
18:                     x, y ← getCoordinates(C_3, i, j)
19:                     for Nonterm ∈ getNonterminals(R, s, f) do
20:                         M_2^Nonterm[x, y] ← {1}
21:     return M_2
```

Из рассмотренного видно, что на каждой итерации основного цикла происходит пересчет шагов, связанных с вычислениями произведения Кронекера и транзитивного замыкания. Хотя алгоритм обладает монотонностью, то есть добавленные дуги, свидетельствующие о существовании пути, никогда не удаляются из графа. То есть в пересчете заново указанных шагов нет никакой необходимости. Таким образом, появляется потребность в улучшении алгоритма построения индекса.

## 1.3. Алгоритм, основанный на матричном умножении

Вторым алгоритмом, основанным на операциях линейной алгебре, является алгоритм, основанный на матричном умножении, предложенный в статье [3]. В качестве входных данных алгоритм принимает грамматику в ОНФХ и граф, который представлен также в виде набора булевых матриц смежности для каждой метки.

Основная идея алгоритма заключается в рассмотрении отдельно правил, где в правой части находится один терминал, вида $A \to a$, и правил, где в правой части находятся два нетерминала, вида $A \to BC$. Для первого типа правил о наличии искомого пути говорит наличие дуги с меткой в виде терминала $a$. Для второго типа правил предлагается умножать матрицы смежности, соответствующие двум нетерминалам $B$ и $C$. Такая операция позволяет соединить дугой вершины, которые достижимы путем, состоящим из 2 дуг, первая из которых с меткой $B$, а вторая — с $C$, тем самым гарантируя обнаружение путей, выводимых из нетерминала $A$.

Было предложено множество модификаций этого алгоритма, покрывающие разнообразные варианты практического использования. Например, в статье Арсения Терехова и др. [2] была предложена модификация, позволяющая искать пути, исходящих из фиксированного набора вершин.

## 1.4. Сравнение алгоритмов, основанных на операциях линейной алгебры

В предыдущих разделах были рассмотрены алгоритм, основанный на произведении Кронекера, и алгоритм, основанный на матричном умножении, реализации которых далее в работе будут обозначены Tns и Mtx, соответственно. Теперь рассмотрим их достоинства и недостатки перед друг другом.

Недостатком алгоритма, основанного на матричном умножении, яв-

ляется представление грамматики в виде ОНФХ, так как это влечёт к увеличению количества правил, что может отрицательно сказаться на производительности. В свою очередь алгоритм, основанный на произведении Кронекера, использует рекурсивный автомат, что является его преимуществом перед аналогом, так как позволяет избежать необходимости преобразовывать исходную грамматику. Однако алгоритм, основанный на произведении Кронекера, оперирует в процессе работы матрицами больших размеров, а именно в результате выполнения произведения Кронекера, взяв матрицу размера $m \times n$ в качестве второго операнда, получается матрица, количество строк и столбцов которой равно количеству строк и столбцов первого операнда увеличенных в $m$ и $n$ раз соответственно.

Таким образом, оба алгоритма основаны на операциях линейной алгебре, что позволяет использовать высокопроизводительные библиотеки при их реализации. Однако алгоритм, основанный на произведении Кронекера, использует рекурсивный автомат, тем самым не увеличивает количество продукций исходной грамматики, что говорит о его конкурентоспособности перед алгоритмом, основанным на матричном умножении.

Но также важным достоинством алгоритма, основанного на матричном умножении, перед аналогом является наличие модификаций для извлечения путей и обнаружения путей, исходящих из фиксированного набора вершиин, так как их существование дает возможность полноценного использования алгоритма, например, в графовых базах данных. Таким образом, появляется потребность в разработке указанных модификаций для алгоритма, основанного на произведении Кронекера.

## 1.5. Библиотека SuiteSparse

Для реализации полученных алгоритмов и последующей оценки их производительности был выбран API GraphBLAS[5], так как на данный мо-

[5]Репозиторий проекта GraphBLAS: https://github.com/GraphBLAS Дата посещения: 25.12.2020

мент является единственным API для работы с графами на языке линейной алгебры.

Реализацию основных концепций API GraphBLAS предоставляют библиотеки GBTL [4], CombBLAS[6], GraphBLAST [12] и SuiteSparse [6]. Для выбора конкретной библиотеки необходимо наличие в ней операций произведения Кронекера, сложения и умножения матриц. Всеми указанными операциями обладают лишь библиотеки GBTL и SuiteSparse. Также достоинством этих библиотек является наличие оберток на языке Python, что облегчает постановку экспериментов. Однако, обертка[7] для GBTL на данный момент не поддерживает операцию произведения Кронекера, что приводит к выбору SuiteSparse и, в частности, обертки pygraphblas[8] для неё.

К достоинствам SuiteSparse также можно отнести использование многопоточности и оптимизаций для разных видов представления векторов и матриц, в том числе учитывается свойство разреженности при хранении и выполнении операций над ними.

Таким образом, для реализации предложенных алгоритмов использовалась библиотека SuiteSparse.

[6]Репозиторий проекта CombBLAS: https://github.com/PASSIONLab/CombBLAS Дата посещения: 25.12.2020

[7]Репозиторий проекта: https://github.com/jessecoleman/gbtl-python-bindings Дата посещения: 25.12.2020

[8]Репозиторий проекта pygraphblas: https://github.com/michelp/pygraphblas Дата посещения: 12.12.2020

# 2. Постановка задачи

Целью данной работы является улучшение производительности алгоритма поиска путей с контекстно-свободными ограничениями, основанного на произведении Кронекера. Для достижения цели были поставлены приведенные ниже задачи.

1. Улучшить алгоритм построения индекса и реализовать полученное улучшение.

2. Разработать алгоритм извлечения путей по результату работы алгоритма построения индекса и реализовать полученный алгоритм.

3. Разработать и реализовать алгоритм обнаружения путей, исходящих из заданного набора стартовых вершин.

4. Провести экспериментальное исследование производительности реализаций. Для демонстрации затраченного времени работы реализованных алгоритмов над исходными данными использовать набор данных *CFPQ_Data*.

# 3. Улучшение построения индекса

Как отмечалось в разделе 1.2 в алгоритме на каждой итерации происходит пересчет произведения Кронекера и транзитивного замыкания результата. Рассмотрим предлагаемое улучшение этих этапов.

Основная идея улучшения заключается в учете лишь тех дуг, которые были добавлены на предыдущем шаге алгоритма (см. листинг 3), для этого введен набор матриц $New\_Nonterm$, где каждая матрица соответствует нетерминалу. Тогда произведение Кронекера рекурсивного автомата и графа вычисляется один раз, а в цикле алгоритма происходит лишь вычисление произведения Кронекера рекурсивного автомата и графа, который состоит из дуг, добавленных на предыдущем шаге. Следующим шагом происходит транзитивное замыкание полученного произведения Кронекера.

Далее, для того, чтобы получить транзитивное замыкание именно произведения Кронекера автомата и исходного графа, без которого шаг обновления графа невозможен, необходимо хранить матрицу $C_3$, полученную на предыдущем шаге алгоритма. Тогда получить нужное транзитивное замыкание можно, соединив дугами все вершины из $C_3$ и $M_3$, которые достигаются путем длины 1, соответственно умножив слева и справа матрицу $C_3$ на матрицу $M_3$ и сложив результаты.

**Listing 3** Улучшенный алгоритм построения индекса

```
1: function contextFreePathQuerying(G, R)
2:     M_1 ← набор матриц смежности для R
3:     M_2 ← набор матриц смежности для G
4:     for s ∈ 0..dim(M_1) − 1 do
5:         for S ∈ getNonterminals(R, s, s) do
6:             for i ∈ 0..dim(M_2) − 1 do
7:                 M_2^S[i, i] ← {1}
8:     C_3 ← M_1 ⊗ M_2                ▷ Вычисление произведения Кронекера
9:     M_3 ← пустая матрица размера dim(C_3)
10:    New_Nonterm ← набор пустых матриц размера dim(M_2)
11:    while Набор M_2 изменяется do
12:        C_3 += M_1 ⊗ New_Nonterms
13:        New_Nonterm ← пустая матрица размера dim(M_2)
14:        C_3 ← transitiveClosure(C_3)
15:        C_3 += M_3 + (M_3 × C_3) × M_3 + M_3 × C_3 + C_3 × M_3
16:        M_3 = C_3
17:        n ← dim(C_3)
18:        for i ∈ 0..n − 1 do
19:            for j ∈ 0..n − 1 do
20:                if C_3[i, j] then
21:                    s, f ← getStates(C_3, i, j)
22:                    x, y ← getCoordinates(C_3, i, j)
23:                    if getNonterminals(R, s, f) ≠ ∅ then
24:                        for Nonterm ∈ getNonterminals(R, s, f) do
25:                            M_2^Nonterm[x, y] ← {1}
26:                            New_Nonterm^Nonterm[x, y] ← {1}
27:    return M_2, C_3
```

# 4. Алгоритм восстановления путей

После вызова функции *contextFreePathQuerying* из листинга 3 в наличии у пользователя оказывается информация о существовании путей между каждой парой вершин в графе. Для фактического же восстановления путей из вершины $v_s$ в вершину $v_f$ с нетерминалом $N$ на дуге $(v_s, v_f)$ предлагаются две функции, представленные на листинге 4. Входной точкой является функция $GetPaths(v_s, v_f, N)$, при вызове которой результатом будет набор путей из $v_s$ в $v_f$, где конкатенация меток каждого пути выводится из нетерминала $N$.

Для восстановления пути из указанной вершины требуется наличие рекурсивного автомата $R$, графа $G$ и результата произведения Кронекера $C_3$, полученные на последней итерации алгоритма обнаружения путей. Тогда, пользуясь тем, что матрица $C_3$ является блочной, происходит пересчет из пары $(q_s^N, v_s)$, характеризующей номер блока по строкам и номер строки в этом блоке соответственно, в $c_s$ и пары $(q_f^N, v_f)$, характеризующей номер блока по столбцам и номер столбца в этом блоке соотвественно, в $c_f$, где $q_s^N$ — начальное состояние автомата для нетерминала N, а $q_f^N$ — одно из конечных состояний автомата для нетерминала N, и вызывается функция $FindPaths(c_s, c_f)$.

Далее предлагается интерпретировать $C_3$ в качестве матрицы смежности графа. В таком случае функция *FindPaths* явно находит путь из $c_s$ в $c_f$, разбив его на две части. Первая часть (*left*) строится из дуг вида $(c_s, k)$, а для получения второй части (*right*) предлагается найти путь из $k$ в $c_f$, вызвав рекурсивно *FindPaths*. Для нахождения всех возможных путей при построении *left* дополнительно проверяется наличие нетерминала на рассматриваемой дуге, и в случае его обнаружения происходит вызов *GetPaths*. Последним этапом соединяются все части искомых путей.

Таким образом, происходит построение всех путей между двумя заданными вершинами графа. Следует отметить, что извлечение путей может выполняться бесконечно долго при наличии циклов в графе. Для реализации предлагается ограничить глубину рекурсивных вызо-

вов, например, количеством запрашиваемых путей или их длиной.

**Listing 4** Алгоритм извлечения путей

```
1:  R ← Рекурсивный автомат
2:  G ← граф
3:  C_3 ← результат произведения Кронекера
4:  size_graph ← количество вершин в графе G
5:  Nontermianls ← набор нетерминалов для R
6:  function GetPaths(v_s, v_f, N)
7:      final_states ← финальные состояния автомата для N
8:      q_s^N ← начальное состояние автомата для N
9:      result_paths ← пустое множество искомых путей
10:     c_s ← q_s^N * size_graph + v_s
11:     for q_f^N ∈ final_states do
12:         c_f ← q_f^N * size_graph + v_f
13:         result_paths = result_paths ∪ FindPaths(c_s, c_f)
14:     return result_paths
15: function FindPaths(c_s, c_f)
16:     if c_s = c_f then
17:         return
18:     result_paths ← пустое множество найденных путей
19:     left ← пустое множество
20:     for k ∈ C_3[c_s] do
21:         q_i ← ⌊c_s/size_graph⌋
22:         q_j ← ⌊k/size_graph⌋
23:         v_s ← c_s  mod  size_graph
24:         v_f ← k  mod  size_graph
25:         labels ← метки на дуге (q_i, q_j)
26:         for label ∈ labels do
27:             if label ∈ Nonterminals then
28:                 left = left ∪ GetPaths(v_s, v_f, label)
29:             else
30:                 left = left ∪ (v_s, label, v_f)
31:         right ← FindPaths(k, c_f)
32:         result_paths = result_paths ∪ left * right
33:     return result_paths
```

# 5. Алгоритм построения индекса с заданным набором стартовых вершин

В процессе работы функции *contextFreePathQuerying* из листинга 3 ищутся пути между всеми вершинами. С практической точки зрения получение такой информации не всегда представляется необходимым, часто пользователю достаточно знать наличие пути из какого-то подмножества вершин. Для решения этой проблемы был разработан алгоритм, представленный на листинге 5. В качестве начальных данных принимаются, как и ранее, рекурсивный автомат и граф, но дополнительно предоставляется набор стартовых вершин графа (*src_ vertices*), то есть только тех вершин, из которых требуется обнаружить пути в любые другие вершины.

В начале (см листинг 5) начальному состоянию автомата для стартового нетерминала присваивается булева матрица, где в каждой ячейке $(i, i)$ находится 1, если $i$ находится во множестве начальных вершин графа. Всем остальным состояниям автоматов ставится в соответствие булевы матрицы, которые будут хранить информацию о текущем наборе стартовых вершин для соответствующего состояния. Далее в основном цикле алгоритма происходит построение графа, с матрицами смежности каждой метки которого и будет вычисляться произведение Кронекера. Для построения такого графа на каждой итерации предлагается следующее.

Взяв автомат для нетерминала, происходит обход всех его состояний, начиная с начального. В процессе обхода берутся все выходящие дуги-переходы (*out_ edges*) из текущего состояния. Для каждой такой дуги матрица смежности исходного графа, соответствующая метке дуги-перехода, умножается на диагональную булеву матрицу, которая сопоставлена состоянию обхода. Тем самым выделяется подграф, который может быть посещен из текущего набора начальных вершин для этого состояния. Информацию о вершинах этого подграфа передается следующему состоянию, а сами подграф становится частью строящегося графа (*Part_ Graph*) и обход продолжается. Тем самым на каждой

итерации алгоритма обнаруживаются только пути, которые исходят из текущего стартового набора вершин графа.

После окончания работы алгоритм вернет набор матриц смежности меток исходного обновленного графа. В таком наборе матрицы для каждого нетерминала рекурсивного автомата $R$ будут содержать *true* на месте *(i,j)*, если из вершины $i$ в вершину $j$ существует искомый путь, при этом гарантируется, что $i$ содержалась в *src_ vertices*.

**Listing 5** Алгоритм построения индекса с набором стартовых вершин

```
1:  function contextFreePathQuerying(G, R, src_vertices:set)
2:      M_1 ← набор матриц смежности для R
3:      M_2 ← набор матрица смежности для G
4:      for s ∈ 0..dim(M_1) − 1 do
5:          for S ∈ getNonterminals(R, s, s) do
6:              for i ∈ 0..dim(M_2) − 1 do
7:                  M_2^S[i, i] ← {1}
8:      boxes ← автоматы для R
9:      start_state ← стартовое состояние для автомата стартового нетерминала
10:     for src_vertex ∈ src_vertices do
11:         Src^{start_state}[src_vertex, src_vertex] ← {1}
12:     while Набор M_2 изменяется или Src_m изменяется do
13:         Part_Graph ← пустой граф
14:         for box ∈ boxes do
15:             states ← состояния для автомата box
16:             for state ∈ states do
17:                 out_edges ← пары вида (label, next state for state)
18:                 for out ∈ out_edges do
19:                     if out.label is nonterminal then
20:                         s_state ← стартовое состояние для boxes[out.label]
21:                         Src^{s_state} += Src^{state}
22:                     Part_Graph^{out.label} += Src^{state} × G^{out.label}
23:                     new_src_vertecies ← get_new_vertices(Part_Graph^{out.label})
24:                     for vertex ∈ new_src_vertecies do
25:                         Src^{out.next_state}[vertex, vertex] ← {1}
26:         M_3 ← M_1 ⊗ Part_Graph
27:         C_3 ← transitiveClosure(M_3)
28:         n ← dim(M_3)
29:         for i ∈ 0..n − 1 do
30:             for j ∈ 0..n − 1 do
31:                 if C_3[i, j] then
32:                     s, f ← getStates(C_3, i, j)
33:                     x, y ← getCoordinates(C_3, i, j)
34:                     for Nonterm ∈ getNonterminals(R, s, f) do
35:                         M_2^{Nonterm}[x, y] ← {1}
36:     return M_2
37: function get_new_vertices(M)
38:     A ← Пустой набор вершиин
39:     for all (v, to) ∈ V^2 | M[v, to] = true do
40:         A ← to
41:     return A
```

# 6. Детали реализаций

Для реализации полученных алгоритмов была разработана архитектура, представленная на рисунке 1.

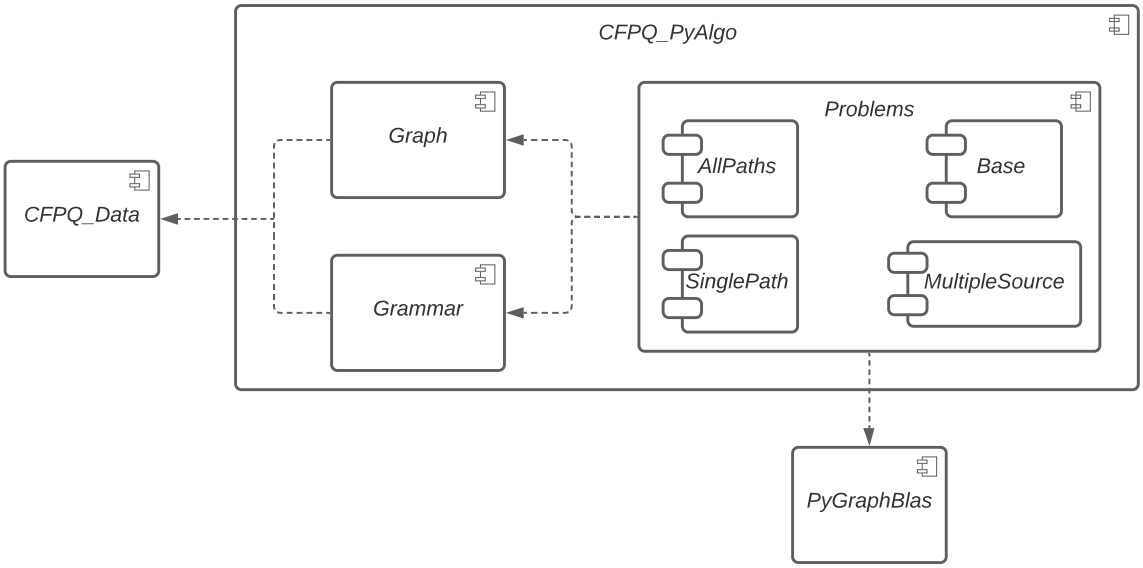


Рис. 1: Архитектура реализаций алгоритмов

Полученная архитектура *CFPQ_PyAlgo* состоит из трёх модулей: *Graph*, *Grammar* и *Problems*.

Модуль *Graph* предназначен для представления графов и их загрузки в качестве входных данных. Модуль *Grammar* реализует хранение и загрузку грамматики в виде ОНФХ, а также рекурсивного автомата. Оба модуля поддерживают данные из *CFPQ_Data*.

Модуль *Problems* разбит на несколько частей. Создание каждой части мотивируется решаемой задачей. Все такие части содержат интерфейс задачи, который реализуют алгоритмы с использованием библиотеки *PyGraphBlas*. В качестве входных данных реализации принимают путь до графа и путь до контекстно-свободной грамматики. Далее каждая конкретная реализация алгоритма выбирает нужные представления графа и грамматики из модулей *Graph* и *Grammar*, соответственно.

Разработанные алгоритмы из разделов 3 и 4 представлены в части *AllPaths*, а из раздела 5 — в *MultipleSource*, исходный код которых на-

ходится в репозитории *CFPQ_PyAlgo*[9]. Для предложенных реализаций написаны тесты на некоторые варианты входных данных и базовая документация. Репозиторий оснащен системой сборки и запуска тестов, а также в нем находятся скрипты для замеров производительности, результаты которых представлены в следующем разделе.

[9]Репозиторий *CFPQ_PyAlgo*: https://github.com/JetBrains-Research/CFPQ_PyAlgo Дата посещения 13.05.2021

# 7. Замеры производительности

Рассмотрим результаты замеров производительности полученных реализаций. Для этого сначала опишем выбранные данные и окружение для постановки экспериментов.

## 7.1. Окружение и данные для экспериментов

Для постановки экспериментов над реализациями был использован ПК с операционной системой Ubuntu 20.04 и конфигурацией: Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz CPU, DDR4 64 Gb RAM.

Из набора данных CFPQ_Data были взяты графы следующих классов:

- RDF — реальные биологические данные;
- MemoryAliases — графы анализа указателей языка Си. Обладают более сложной структурой в отличии от графов класса RDF.

Таким образом были выбраны графы, представленные в таблице 1.

| Graph | #V | #E |
|---|---|---|
| eclass_514en | 239 111 | 523 727 |
| enzyme | 48 815 | 109 695 |
| geospecies | 450 609 | 2 201 532 |
| go | 272 770 | 534 311 |
| go-hierarchy | 45 007 | 980 218 |
| taxonomy | 5 728 398 | 14 922 125 |
| arch | 3 448 422 | 5 940 484 |
| crypto | 3 464 970 | 5 976 774 |
| drivers | 4 273 803 | 7 415 538 |
| fs | 4 177 416 | 7 218 746 |

Таблица 1: Графы

В качестве грамматик были взяты следующие:

- $G_1 : S \to \overline{sco}\ \ S\ \ sco\ \mid\ \overline{type}\ \ S\ \ type\ \mid\ \overline{sco}\ \ sco\ \mid\ \overline{type}\ \ type$
- $G_2 : S \to \overline{sco}\ \ S\ \ sco\ \mid\ sco$
- $Geo : S \to bt\ \ S\ \ \overline{bt}\ \mid\ bt\ \ \overline{bt}$
- $MA$ :

$$S \to \overline{d}\ \ V\ \ d$$
$$V \to (S?\overline{a})^* S?(aS?)^*$$

Где $\overline{x}$ обозначает ребро обратное ребру с меткой $x$ в графе. Количество терминалов из выбранных грамматик, соответствующие меткам в графах, приведено в таблице 2.

| Graph | #sco | #type | #bt | #a | #d |
|---|---|---|---|---|---|
| eclass_514en | 90 512 | 72 517 | — | — | — |
| enzyme | 8 163 | 14 989 | — | — | — |
| geospecies | 0 | 89 062 | 20 867 | — | — |
| go | 90 512 | 58 483 | — | — | — |
| go-hierarchy | 490 109 | 0 | — | — | — |
| taxonomy | 2 112 637 | 2 508 635 | — | — | — |
| arch | — | — | — | 671 295 | 2 298 947 |
| crypto | — | — | — | 678 408 | 2 309 979 |
| drivers | — | — | — | 858 568 | 2 849 201 |
| fs | — | — | — | 824 430 | 2 784 943 |

Таблица 2: Количество терминалов из выбранных грамматик, соответствующих меткам в графах

## 7.2. Эксперименты над улучшением алгоритма построения индекса

Сравнения реализации улучшения, предложенного в разделе 3, производились с реализацией базового алгоритма, основанного на произве-

дении Кронекера, использующей тот же набор технологий и ПК для проведения экспериментов.

В ходе проведения экспериментов было произведено 5 запусков с каждым графом и каждой грамматикой. В конце бралось среднее значение из каждой выборки. Таким образом результаты измерений представлены с $\pm\Delta = \pm 0,06$ сек точностью оценки при доверительной вероятности 0.95.

| Name | $G_1$ | | $G_2$ | | *Geo* | | *MA* | |
|---|---|---|---|---|---|---|---|---|
| | Tns2 | Tns1 | Tns2 | Tns1 | Tns2 | Tns1 | Tns2 | Tns1 |
| eclass_514en | 0.24 | 0.25 | 0.25 | 0.26 | — | — | — | — |
| enzyme | 0.03 | 0.04 | 0.02 | 0.04 | — | — | — | — |
| geospecies | 0.08 | 0.09 | $< 0.01$ | 0.01 | 26.12 | 34.12 | — | — |
| go-hierarchy | 0.16 | 0.19 | 0.23 | 0.29 | — | — | — | — |
| go | 1.56 | 1.68 | 1.21 | 1.37 | — | — | — | — |
| pathways | 0.01 | 0.02 | 0.01 | 0.01 | — | — | — | — |
| taxonomy | 4.81 | 5.37 | 3.75 | 3.81 | — | — | — | — |
| arch | — | — | — | — | — | — | 262.45 | 390.05 |
| crypto | — | — | — | — | — | — | 267.52 | 395.98 |
| drivers | — | — | — | — | — | — | 1309.57 | 2114.16 |
| fs | — | — | — | — | — | — | 470.49 | 745.97 |

Таблица 3: Результаты экспериментов над улучшением построения индекса, время в секундах

В таблице 3 продемонстрировано время в секундах работы алгоритмов над графами в зависимости от грамматик. Знак ”—” означает неприменимость грамматики к графу. Tns2 — результаты алгоритма с улучшением, Tns1 — результаты алгоритма без улучшения.

Таким образом, наблюдается всюду положительный результат. А именно, для графов типа RDF уменьшение времени выполнения составило до 23%, а для графов MemoryAliases — до 38%. Особо выразительный выигрыш алгоритма с улучшением получился на графах типа MemoryAliases, так как в виду их сложной структуры происходит

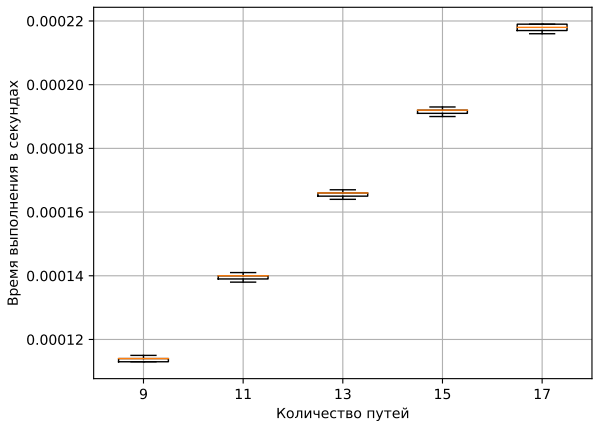


Рис. 2: Результаты эксперимента над извлечением путей Tns с ограничением по длине 20

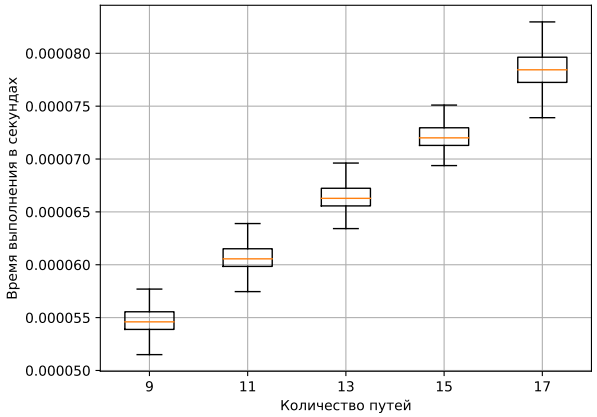


Рис. 3: Результаты эксперимента над извлечением путей Mtx с ограничением по длине 20

большое число действий при транзитивном замыкании в базовой версии алгоритма, а предложенное улучшение позволяет уменьшить число таких операций.

## 7.3. Эксперименты над алгоритмом извлечения путей

Для реализации предложенного алгоритма извлечения путей в разделе 4 было выбрано ограничение на глубину рекурсии в виде длины пути. Таким образом, к процедуре *GetPaths* добавляется ещё один аргумент, который характеризует ограничение на длину пути, и в таком случае результатом будет набор путей длины меньше заданной этим аргументом.

В ходе эксперимента был выбран из таблицы 1 граф *eclass_514en* и грамматика $G_1$ и происходило восстановление путей из всех пар вершин, между которыми был обнаружен путь алгоритмом построения индекса, с ограничением на длину в 20 и 50. Сравнение происходит с алгоритмом, основанным на матричном умножении.

Таким образом были получены результаты, изображенные на рисунках 2 и 4, для предложенного алгоритма.

На графиках приведены распределение времени для пяти самых ча-

сто встречающихся количеств найденных путей. Аналогичные результаты для алгоритма, основанного на матричном умножении, приведены на рисунках 3 и 5.

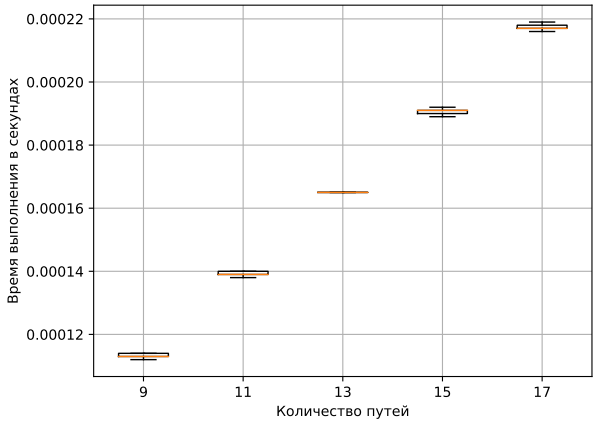


Рис. 4: Результаты эксперимента над извлечением путей Tns с ограничением по длине 50

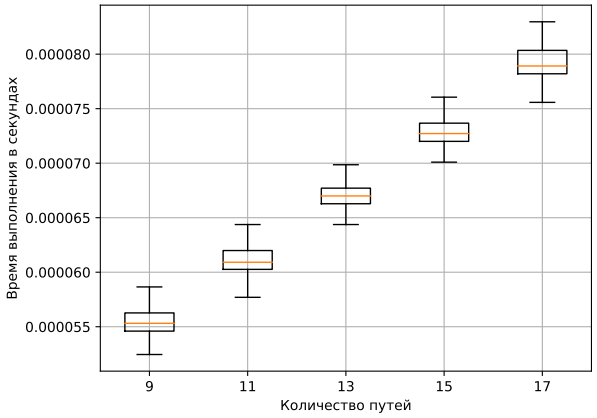


Рис. 5: Результаты эксперимента над извлечением путей Mtx с ограничением по длине 50

На рисунках 2 и 4 наблюдается незначительный прирост времени выполнения при увеличении ограничения на длину путей.

Таким образом предложенный алгоритм медленнее аналога более чем в 100 раз по времени выполнения, так как в процессе обнаружения путей алгоритм, основанный на матричном умножении, строит структуру, которая явно хранит вершины, являющиеся частями искомых путей, в то время как для алгоритма, основанного на произведении Кронекера, приходится обходить матрицу смежности $C_3$.

## 7.4. Эксперименты над алгоритмом обнаружения путей с заданным набором стартовых вершин

Для постановки экспериментов над реализацией предложенного алгоритма в разделе 5 были выбраны из таблицы 1 графы *go-hierarchy* и *eclass_514en* и грамматика $G_1$. Для каждого графа множество его вершин было разбито на непересекающиеся множества фиксированного размера. Далее производился запуск для каждого такого множества в качестве стартового набора вершин. Полученные результаты пред-

ставлены на рисунках 6 и 7 синим цветом. Штрихом на диаграммах представлена медиана.

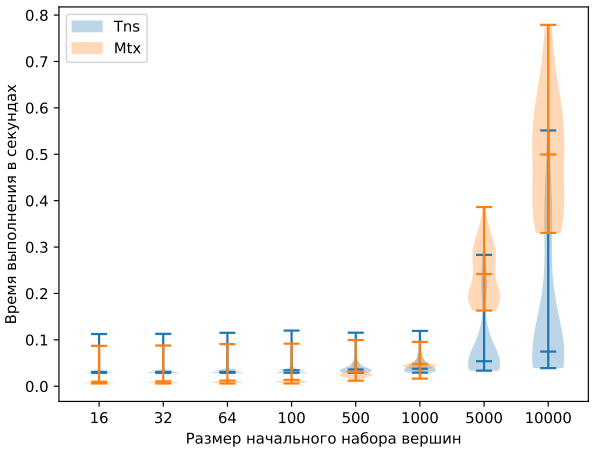


Рис. 6: Результаты экспериментов над алгоритмом с начальном мновом вершин на графе *eclass_514en*

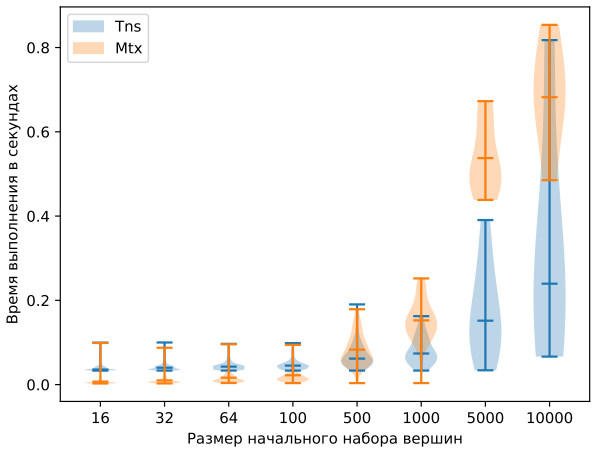


Рис. 7: Результаты экспериментов над алгоритмом с начальном мновом вершин на графе *go-hierarchy*

Сравнение производилось с аналогичным алгоритмом через матричное умножение. Результат работы аналога представлен на рисунках оранжевым цветом.

Таким образом для взятых графов алгоритм через произведение Кронекера, сравним с аналогом на множествах начальных вершин размером не превосходящих 1000. Начиная с размера начального множества вершин 5000, алгоритм через произведение Кронекера демонстрирует меньшее время работы. Однако видно, что цель создания алгоритма с точки зрения времени выполнения оправдывается только при размере начального множества вершин меньше 1000. Так как в таблице 3 указано, что полную информацию о достижимости для графа *eclass_514en* можно получить за 0.24 с., а *go-hierarchy* — за 0.16 с.

# Заключение

В ходе работы были выполнены следующие задачи.

1. Улучшен алгоритм построения индекса с использованием динамического пересчета некоторых шагов алгоритма и реализовано полученное улучшение.

2. Разработан и реализован алгоритм извлечения путей по результату работы алгоритма построения индекса.

3. Разработан и реализован алгоритм построения индекса с фиксированным набором стартовых вершин.

4. Произведено экспериментальное исследование производительности реализаций. Полученное улучшение алгоритма поиска путей демонстрирует уменьшение времени до 38 % по сравнению с версией без улучшения. Разработанный алгоритм с набором стартовых вершин быстрее существующего аналога. Однако алгоритм извлечения путей оказался медленнее до 100 раз того же аналога ввиду более сложно устроенных структур необходимых в процессе его работы.

Также результат был изложен в статье "Context-Free Path Querying with All-Path Semantics by Matrix Multiplication", которая принята на конференцию GRADES-NDA 2021.

Реализации представлены в репозитории: https://github.com/JetBrains-Research/CFPQ_PyAlgo.

# Список литературы


[1] Analysis of Recursive State Machines / Rajeev Alur, Michael Benedikt, Kousha Etessami et al. // ACM Trans. Program. Lang. Syst. — 2005. — Vol. 27, no. 4. — P. 786–818. — URL: https://doi.org/10.1145/1075382.1075387.

[2] Arseniy Terekhov Vlada Pogozhelskaya Vadim Abzalov Timur Zinnatulin Semyon Grigorev. Multiple-Source Context-Free Path Querying in Terms of Linear Algebra // Advances in Database Technology – EDBT 2021. — 2021. — P. 487–492. — URL: http://dx.doi.org/10.5441/002/edbt.2021.56.

[3] Azimov Rustam, Grigorev Semyon. Context-Free Path Querying by Matrix Multiplication // Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). — GRADES-NDA '18. — New York, NY, USA : Association for Computing Machinery, 2018. — 10 p. — URL: https://doi.org/10.1145/3210259.3210264.

[4] Buluç Aydın, Gilbert John R. The Combinatorial BLAS: design, implementation, and applications // The International Journal of High Performance Computing Applications. — 2011. — Vol. 25, no. 4. — P. 496–509. — https://doi.org/10.1177/1094342011403516.

[5] Context-Free Path Querying by Kronecker Product / Egor Orachev, Ilya Epelbaum, Rustam Azimov, Semyon Grigorev // Advances in Databases and Information Systems / Ed. by Jérôme Darmont, Boris Novikov, Robert Wrembel. — Cham : Springer International Publishing, 2020. — P. 49–59.

[6] Davis Timothy A., Aznaveh Mohsen, Kolodziej Scott. Write Quick, Run Fast: Sparse Deep Neural Network in 20 Minutes of Development Time via SuiteSparse:GraphBLAS // 2019 IEEE High Performance Extreme Computing Conference (HPEC). — 2019. — P. 1–6.

[7] Evaluation of the Context-Free Path Querying Algorithm Based on Matrix Multiplication / Nikita Mishin, Iaroslav Sokolov, Egor Spirin et al. // Proceedings of the 2Nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). — GRADES-NDA'19. — New York, NY, USA : ACM, 2019. — P. 12:1–12:5. — URL: http://doi.acm.org/10.1145/3327964.3328503.

[8] Hellings J. Path Results for Context-free Grammar Queries on Graphs // ArXiv. — 2015. — Vol. abs/1502.02242.

[9] Reps Thomas. Program Analysis via Graph Reachability // Proceedings of the 1997 International Symposium on Logic Programming. — ILPS '97. — Cambridge, MA, USA : MIT Press, 1997. — P. 5–19. — URL: http://dl.acm.org/citation.cfm?id=271338.271343.

[10] 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.

[11] Sevon Petteri, Eronen Lauri. Subgraph queries by context-free grammars // Journal of Integrative Bioinformatics : JIB. — 2008. — Vol. 5, no. 100, 16 s.

[12] Yang Carl, Buluc Aydin, Owens John D. GraphBLAST: A High-Performance Linear Algebra-based Graph Framework on the GPU. — 2020. — 1908.01407.

[13] 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 2019 / Ed. by Tanu Malik, Carlos Maltzahn, Ivo Jimenez. — United States : Association for Computing Machinery, Inc, 2019. — 7. — P. 121–132.