Санкт-Петербургский государственный университет

Кутуев Владимир Александрович

Выпускная квалификационная работа

Экспериментальное исследование алгоритмов контекстно-свободной достижимости применительно к задачам статического анализа кода

Уровень образования: магистратура

Направление *09.04.04 «Программная инженерия»*

Основная образовательная программа *ВМ.5666.2021 «Программная инженерия»*

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

Рецензент:
старший преподаватель, Санкт-Петербургский Политехнический Университет Петра Великого М. А. Беляев

Санкт-Петербург
2023

Saint Petersburg State University

Vladimir Kutuev

Master's Thesis

Experimental study of context-free-language reachability algorithms as applied to static code analysis

Education level: master

Speciality *09.04.04 «Software Engineering»*

Programme *BM.5666.2021 «Software Engineering»*

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

Reviewer:
Senior Lecturer, Peter the Great St.Petersburg Polytechnic University M. A. Belyaev

Saint Petersburg
2023

Оглавление

Введение

Статический анализ играет важную роль в задаче поиска ошибок в коде. Однако многие из методов статического анализа основаны на сопоставлении участков кода с некоторым шаблоном: если участок кода соответствует шаблону, считается, что он содержит ошибку. Такой подход может приводить как к пропуску существующих ошибок, так и выдаче ложных предупреждений. Алгоритмы статического анализа, которые выделяют ошибки на основе различных свойств программы, например, контекстов вызова и потока управления, позволяют обнаруживать больше истинных ошибок и сообщать меньше ложных предупреждений.

Многие виды статического анализа могут быть сформулированы как задача контекстно-свободной (КС) достижимости в графе [14, 11, 12, 10]. Один из примеров — анализ псевдонимов (анализ указателей). Он позволяет обнаруживать использование освобождённой памяти, взаимные блокировки и обращение к выделенной памяти через тип с несоответствующим размером [5]. Ещё один пример — Points-to анализ, учитывающий поля, особенностью которого являются большие грамматики, используемые для анализа.

Существует множество алгоритмов, решающих задачу КС-достижимости [8, 13, 6, 2]. Среди них можно выделить алгоритмы, основанные на операциях линейной алгебры, так как такие операции хорошо поддаются распараллеливанию. В исследовании Никиты Мишина и др. [4] был произведён сравнительный анализ времени работы нескольких реализаций матричного алгоритма Рустама Азимова [2], основанных на различных специализированных матричных библиотеках. Однако графы, на которых проводилось исследование, значительно меньше получаемых по исходному коду для статического анализа.

В данной работе предлагается изучить различные алгоритмы КС-достижимости в графах и сравнить их производительность на больших графах, полученных по реальным программам.

1. Постановка задачи

Целью данной работы является экспериментальное исследование алгоритмов КС-достижимости в задаче статического анализа кода.

Для достижения цели были поставлены следующие задачи.

1. Рассмотреть возможность применения алгоритмов, основанных на операциях линейной алгебры, для Points-to анализа, учитывающего поля, и предложить модификацию алгоритма, подходящую для этого анализа.

2. Оптимизировать реализации алгоритмов, основанных на операциях линейной алгебры.

3. Провести замеры производительности алгоритмов, основанных на операциях линейной алгебры, на графах, полученных по реальным программам, сравнить их с другими алгоритмами КС-достижимости.

2. Обзор

В данном разделе определены основные необходимые понятия из теории формальных языков, рассмотрены способы сведения анализа псевдонимов и Points-to анализа, учитывающего поля, к задаче КС-достижимости, приведены решающие её алгоритмы, основанные на операциях линейной алгебры. Также приведены различные реализации, решающие задачу КС-достижимости.

2.1. Терминология

Контекстно-свободная грамматика — $G = \langle \Sigma, N, P, S \rangle$, где:

- Σ — множество терминальных символов;
- N — множество нетерминальных символов;
- P — множество продукций, каждая продукция имеет вид $A \to \alpha$, где $A \in N, \alpha \in (\Sigma \cup N)^* \cup \varepsilon$;
- $S \in N$ — стартовый нетерминал.

Последовательность терминалов и нетерминалов $\gamma\alpha\delta$ **непосредственно выводится из** $\gamma\beta\delta$ *при помощи правила* $\alpha \to \beta$ ($\gamma\alpha\delta \Rightarrow \gamma\beta\delta$), если:

- $\alpha \to \beta \in P$;
- $\gamma, \delta \in \{\Sigma \cup N\}^* \cup \varepsilon$.

Отношение выводимости является рефлексивно-транзитивным замыканием отношения непосредственной выводимости. $\alpha \overset{*}{\Rightarrow} \beta$ означает $\exists \gamma_0, \dots \gamma_k : \ \alpha \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_{k-1} \Rightarrow \gamma_k \Rightarrow \beta$.

Язык, задаваемый грамматикой — множество строк, из стартового нетерминала грамматики $\mathcal{L}(G) = \{\omega \in \Sigma^* \mid S \overset{*}{\Rightarrow} \omega\}$.

Контекстно-свободная грамматика находится в **ослабленной нормальной форме Хомского (ОНФХ)**, если все продукции имеют вид:

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

ОНФХ отличается от нормальной формы Хомского тем, что в ней допускаются продукции вида $A \to \varepsilon$ для любого нетерминала, а не только для стартового.

Ориентированный граф с метками на ребрах $\mathcal{G} = \langle V, E, L \rangle$ есть тройка объектов, где V — конечное непустое множество вершин, $E \subseteq V \times L \times V$ — конечное множество рёбер, L — конечное множество меток графа. Здесь и далее считается, что вершины графа индексируются целыми числами, то есть $V = \{0\ , ...,\ |V| - 1\}$.

Путь π в графе $\mathcal{G} = \langle V, E, L \rangle$ — это последовательность рёбер e_0, e_1, e_{n-1}, где $e_i = (v_i, l_i, u_i) \in E$, для любых $e_i, e_{i+1} : u_i = v_{i+1}$. Путь между вершинами v и u будем обозначать как $v\pi u$. Путь $\pi = (v_0, l_0, v_1), ..., (v_{n-1}, l_{n-1}, v_n)$, формирует слово $\omega(\pi) = l_0 ... l_{n-1}$.

Рекурсивный автомат [1] над конечным алфавитом Σ есть $R = \langle M, m, \{C_i\}_{i \in M} \rangle$, где:

- M — конечное множество меток;
- $m \in M$ — начальная метка;
- $\{C_i\}_{i \in M}$ — множество *конечных автоматов*, где $C_i = \langle \Sigma \cup M, Q_i, q_i^0, F_i, \delta_i \rangle$:
  - $\Sigma \cup M$ — 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 $F_i \subseteq Q_i$;
  - δ_i — функция переходов C_i, где $\delta_i : Q_i \times (\Sigma \cup M) \to Q_i$.

Для ориентированного графа $\mathcal{G}$ и КС языка $\mathcal{L}$ задача **контекстно-свободной достижимости** заключается в поиске всех таких пар вершин (v, u), что между ними существует путь $v\pi u$ такой, что $\omega(\pi) \in \mathcal{L}$. Результат запроса обозначается как $R = \{(v, u) \mid \exists v\pi u : \omega(\pi) \in \mathcal{L}\}$.

2.2. Анализ псевдонимов

Два указателя являются псевдонимами, если они указывают на одну и ту же область памяти. Задача поиска псевдонимов, нечувствительная к потоку данных, может быть выражена как задача контекстно-свободной достижимости на графе выражений программы [15]. В этом графе вершинам соответствуют переменная-указатель x, разыменование указателя $*x$ и взятие адреса указателя $\&x$. Оператор присваивания с этими выражениями порождает рёбра по следующим правилам:

Выражение	Ребро в графе
x = y	$x \xleftarrow{a} y$
*x = y	$*x \xleftarrow{a} y$
x = *y	$x \xleftarrow{a} *y$
x = &y	$x \xleftarrow{a} \&y$

Каждая операция выделения памяти ($x = malloc(...)$) обрабатывается аналогично взятию адреса, то есть в граф добавляется ребро от выделенного участка памяти к ячейке памяти, в которую записывается указатель $x \xleftarrow{a} \&O$, где $\&O$ — адрес выделенного участка памяти. Также для каждого указателя x добавляются рёбра разыменования с меткой d: $x \xrightarrow{d} *x$ и $\&x \xrightarrow{d} x$.

Если достроить полученный граф обратными рёбрами: для каждого ребра $x \xrightarrow{a} y$ добавляется $x \xleftarrow{\overline{a}} y$, а для ребра $x \xrightarrow{d} y$ добавляется $x \xleftarrow{\overline{d}} y$, то такое представление программы позволяет сформулировать задачу анализа псевдонимов как задачу поиска путей с контекстно-свободными ограничениями, задаваемую грамматикой с алфавитом $\Sigma = \{a, \overline{a}, d, \overline{d}\}$, нетерминалами $N = \{MA, VA\}$, стартовым нетерминалом MA и следующими продукциями (продукция для нетерминала VA для краткости

Программа

$v1 = \&v2;$
$v3 = \&v1;$
$v4 = malloc(...);$
$*v2 = v4;$
$v5 = *v3;$
$v6 = *v5;$

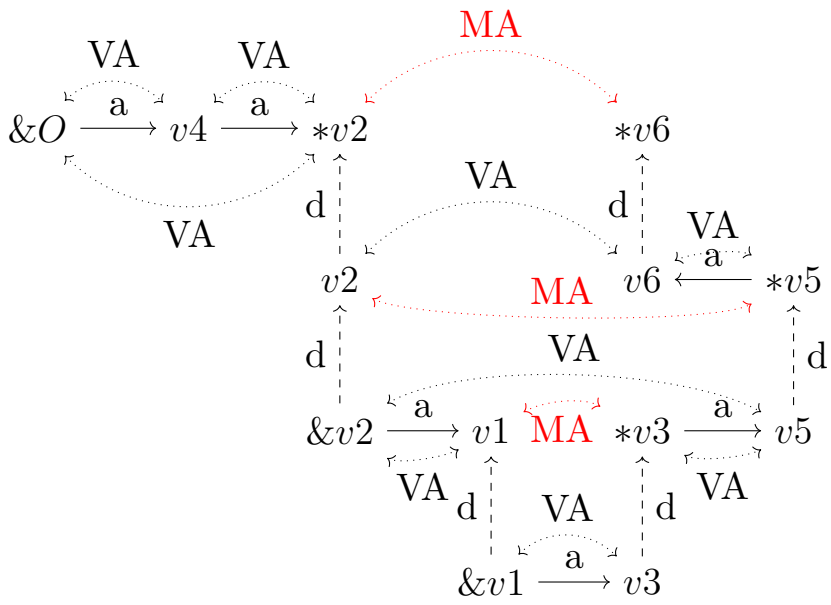


Рис. 1: Анализ псевдонимов. Программа и её граф выражений

записана в формате, отличном от описанного выше, но может быть сведена к нему):

$$\begin{array}{ll} \text{Memory alias} & MA \to \overline{d}\ VA\ d \\ \text{Value alias} & VA \to (MA?\ \overline{a})^*\ MA?\ (a\ MA?)^* \end{array}$$

На рис. 1 представлены пример программы и построенный по ней граф выражений. Также этот граф дополнен рёбрами, которые показывают контекстно-свободную достижимость, задаваемую грамматикой. Если в графе между вершинами v_1 и v_2 существует ребро MA, то выражения в коде, соответствующие этим вершинам, являются псевдонимами.

Кроме задачи поиска всех пар псевдонимов можно выделить такие задачи, как проверка того, что два выражения являются псевдонимами, и поиск всех псевдонимов для выбранных выражений. Их можно переформулировать как задачу поиска путей с контекстно-свободными ограничениями с заданным множеством исходных вершин над тем же графом. Это позволит ускорить анализ, отказавшись от обработки неинтересующих вершин.

2.3. Points-to анализ, учитывающий поля

Ещё один вид анализа, который может быть сформулирован как задача поиска путей с контекстно-свободными ограничениями — Points-to анализ, учитывающий поля, применяемый для анализа Java-программ. Он заключается в определении, на какие объекты кучи могли указывать переменные в процессе работы программы.

По исходной программе строится граф, вершины которого соответствуют объектам на куче и переменным программы, а рёбра добавляются на основании присваиваний по следующим правилам:

Выражение	Ребро в графе
x = new Obj();	$x \xrightarrow{alloc} h$
x = y;	$x \xrightarrow{assign} y$
x = y.f;	$x \xrightarrow{load_f} y$
x.f = y;	$x \xrightarrow{store_f} y$

Если достроить полученный граф обратными рёбрами: для каждого ребра $x \xrightarrow{label} y$ добавляется $x \xleftarrow{\overline{label}} y$, то такое представление программы позволяет сформулировать задачу анализа указателей как задачу поиска путей с контекстно-свободными ограничениями, задаваемую контекстно-свободной грамматикой:

$$\begin{array}{l} PointsTo \rightarrow (assign \mid load_f\ Alias\ store_f)^*\ alloc \\ Alias \rightarrow PointsTo\ \ \overline{FlowsTo} \qquad\qquad \forall\ \mathrm{f} \in Fields \\ FlowsTo \rightarrow \overline{alloc}\ (\overline{assign} \mid \overline{store_f}\ Alias\ \overline{load_f})^* \end{array}$$

На самом деле, это не КС-грамматика, а шаблон грамматики. Каждая программа обладает уникальным набором полей, поэтому грамматика, с помощью которой будет производиться анализ, будет получена из данного шаблона и графа, построенного по конкретной программе. Так, на рис. 2 представлен пример программы с тремя различными полями, построенный по ней граф выражений, дополненный рёбрами, которые показывают контекстно-свободную достижимость, задаваемую грамматикой.

Программа

$v1 = new\ Obj();\ //\ h1$
$v2 = new\ Obj();\ //\ h2$
$v4 = new\ Obj();\ //\ h3$
$v6 = new\ Obj();\ //\ h4$
$v5 = v4;$
$v5 = v6;$
$v1.h = v1;$
$v2.g = v1;$
$v4.f = v2;$
$v7 = v5.f;$
$v9 = v6.f;$
$v3 = v2.g;$
$v8 = v7.g;$
$v10 = v9.g;$
$v10 = v10.h;$

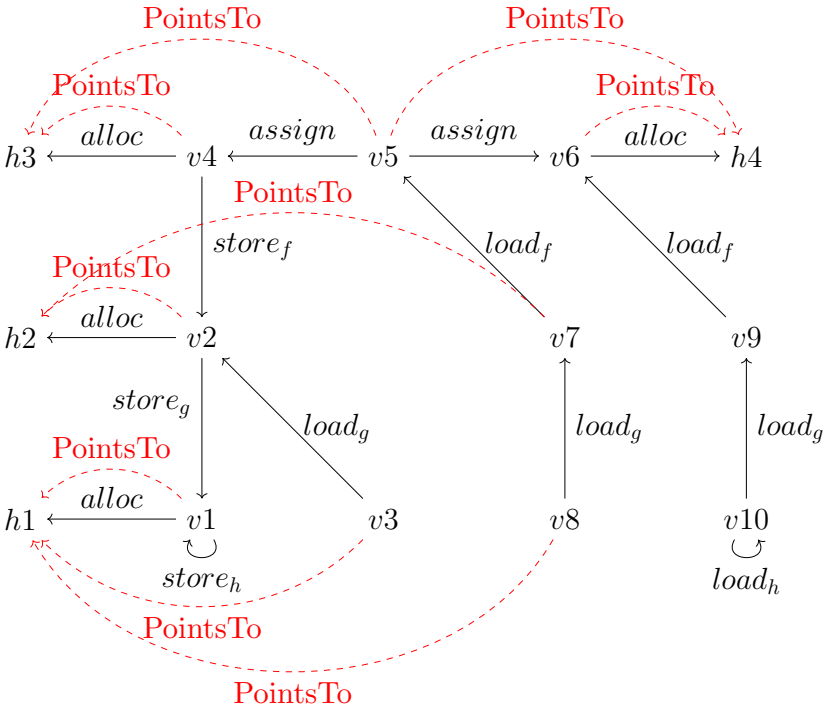


Рис. 2: Points-to анализ, учитывающий поля. Программа и её граф выражений

2.4. Алгоритмы КС-достижимости, основанные на операциях линейной алгебры

Далее будут рассмотрены алгоритмы, решающие задачу контекстно-свободной достижимости, основанные на операциях линейной алгебры: матричном умножении и произведении Кронекера.

2.4.1. Матричный алгоритм

В статье [2] был преложен алгоритм, основанный на умножении матриц. На вход алгоритму подаётся граф $\mathcal{G} = \langle V, E, l \rangle$ и контекстно-свободная грамматика $G = \langle \Sigma, N, P, S \rangle$ в ослабленной нормальной форме Хомского.

Для каждого нетерминала $A \in N$ создаётся матрица смежности M^A. Если $M^A[i, j] = true$, то в графе между вершинами с номерами i и j существует путь, конкатенация меток на рёбрах которого образует слово, выводимое из нетерминала A в заданной грамматике. Инициализация

Algorithm 1 Матричный алгоритм Рустама Азимова

```
1: function MatrixCFPQ(𝒢 = (V, E, l), G = (Σ, N, P, S))
2:     n ← |V|
3:     M ← {M^A | A ∈ N, M^A[i, j] ← false, for all i, j}
4:     for all v ∈ V do                                  ▷ Инициализация матриц
5:         for all A → ε do
6:             M^A[v, v] ← true
7:         end for
8:     end for
9:     for all (v, u) ∈ E do
10:        label ← l((v, u))
11:        for all A → label ∈ P do
12:            M^A[u, v] ← true
13:        end for
14:    end for
15:    while ∃A : M^A is changing do                     ▷ Вычисление замыкания
16:        for all A → BC ∈ P do
17:            M^A ← M^A + M^B · M^C
18:        end for
19:    end while
20:    return M^S
21: end function
```

матриц происходит с помощью ε-правил: для правил вида $A \to \varepsilon$ для всех вершин добавляется петля $M^A[i, i] = true$, и простых правил грамматики (правил вида $A \to a$): если в графе между вершинами i и j есть ребро с меткой a, то $M^A[i, j] = true$.

Затем происходит поиск транзитивного замыкания (строки 15–19 алгоритма 1), для правил вида $A \to BC$ матрица смежности нетерминала A обновляется следующим образом: $M^A = M^A + M^B \cdot M^C$. Алгоритм использует булевы матрицы, поэтому в качестве сложения её элементов используется дизъюнкция, а в качестве умножения — конъюнкция.

Результатом работы алгоритма является матрица смежности для стартового нетерминала — M^S. Эта матрица будет содержать в себе информацию о всех парах вершин, соединённых путями, метки на рёбрах которых образуют слово из языка грамматики.

2.4.2. Тензорный алгоритм

Ещё один алгоритм КС-достижимости, основанный на операциях линейной алгебры — алгоритм, основанный на произведении Кронеке-

ра [3]. Алгоритм (листинг 2) принимает граф $\mathcal{G}$ и рекурсивный автомат R. Его идея заключается в использовании модификации алгоритма пересечения конечных автоматов [7] для пересечения рекурсивного автомата и графа. Для этого граф $\mathcal{G}$ рассматривается как автомат: вершины считаются состояниями, а дуги — переходами.

Входной граф и рекурсивный автомат представляются в виде композиции булевых матриц смежности для всех меток на рёбрах графа и переходах автомата. Пересечение выполняется с использованием произведения Кронекера, а множество рекурсивных вызовов учитывается с помощью транзитивного замыкания.

После инициализации матриц смежности проверяется наличие состояний в автомате R, которые одновременно являются стартовыми и конечными, и при наличии таковых для каждой вершины графа добавляется петля с соответствующей меткой.

Основной цикл алгоритма выполняется, пока набор матриц смежности для графа меняется. Первым шагом цикла происходит вычисление произведения Кронекера, тем самым создаётся матрица смежности нового автомата. Далее результат транзитивно замыкается для получения информации о достижимости состояний в полученном автомате. И последним шагом происходит обновление матрицы смежности графа.

2.4.3. Инкрементальная версия тензорного алгоритма

В работе [9] предложена следующая оптимизация тензорного алгоритма. Чтобы вычислять произведение Кронекера инкрементально, можно воспользоваться левой дистрибутивностью данной операции. Пусть A_2 — матрица, содержащая элементы, добавленные на последней итерации, B_2 — матрица, содержащая элементы, добавленные ранее. Тогда в строке 13 алгоритма 2 по левой дистрибутивности $M_1 \otimes M_2 = M_1 \otimes (A_2 + B_2) = M_1 \otimes A_2 + M_1 \otimes B_2$. Однако $M_1 \otimes B_2$ уже был посчитан на предыдущей итерации, и его транзитивное замыкание хранится в M_3. Таким образом, остаётся обновить некоторые элементы M_3, вычислив $M_1 \otimes A_2$.

Algorithm 2 Алгоритм, основанный на произведении Кронекера

```
1: function contextFreePathQuerying(𝒢, R)
2:     M_1 ← набор матриц смежности для R                ▷ Инициализация матриц
3:     M_2 ← набор матриц смежности для 𝒢
4:     n ← dim(M_1)×dim(M_2)
5:     for s ∈ 0..dim(ℳ_1) − 1 do
6:         for S ∈ getNonterminals(R, s, s) do
7:             for i ∈ 0..dim(ℳ_2) − 1 do
8:                 M_2^S[i, i] ← {1}
9:             end for
10:        end for
11:    end for
12:    while M_2 is changing do                          ▷ Тело алгоритма
13:        M_3 ← M_1 ⊗ M_2                               ▷ Произведение Кронекера
14:        C_3 ← transitiveClosure(M_3)
15:        for i ∈ 0..n − 1, 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:                end for
22:            end if
23:        end for
24:    end while
25:    return M_2
26: end function
27: function getStates(M_1, i, j)   ▷ Получение номеров состояний автомата по
    номерам ячеек в матрице
28:    r ← dim(M_1)
29:    return ⌊i/r⌋ , ⌊j/r⌋
30: end function
31: function getCoordinates(M_2, i, j)   ▷ Получение номеров вершин графа по
    индексам ячеек в матрице
32:    n ← dim(M_2)
33:    return i mod n, j mod n
34: end function
```

2.5. Исследуемые реализации

Ниже перечислены различные реализации, которые могут выполнять описанные виды статического анализа, сформулированные как задача КС-достижимости.

- CFPQ_PyAlgo[1] — платформа для разработки и тестирования алгоритмов запросов к графам с контекстно-свободными ограничениями, основанных на операциях линейной алгебры, содержащая реализации для:
  - CPU: для выполнения операций с разреженными матрицами используется pygraphblas[2] — python-обёртка над библиотекой SuiteSparse[3];
  - GPU: для выполнения операций с разреженными булевыми матрицами используется библиотека cuBool[4].
- GLL4Graph[5] — реализация алгоритма КС-достижимости, основанная на алгоритме синтаксического анализа GLL, написанная на Java. Поддерживает хранение графа в оперативной памяти и в графовой базе данных Neo4j.
- Graspan[6] — высокопроизводительная система межпроцедурного статического анализа.
- Gigascale[7] — высокопроизводительный инструмент для учитывающего поля Points-to анализа Java-программ.

[1]Репозиторий CFPQ_PyAlgo: https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_PyAlgo, дата посещения 10.05.2023

[2]Репозиторий pygraphblas: https://github.com/Graphegon/pygraphblas, дата посещения 22.05.2023

[3]Репозиторий SuiteSparse: https://github.com/DrTimothyAldenDavis/GraphBLAS, дата посещения 22.05.2023

[4]Репозиторий cuBool: https://github.com/SparseLinearAlgebra/cuBool, дата посещения 22.05.2023

[5]Репозиторий GLL4Graph: https://github.com/FormalLanguageConstrainedPathQuerying/GLL4Graph, дата посещения 22.05.2023

[6]Репозиторий Graspan-C: https://github.com/Graspan/Graspan-C Дата посещения 10.05.2023

[7]Репозиторий Gigascale: https://bitbucket.org/jensdietrich/gigascale-pointsto-oopsla2015/src/master/, дата посещения 22.05.2023

3. Адаптация тензорного алгоритма для Points-to анализа, учитывающего поля

Для практического применения алгоритма, основанного на умножении матриц, желательно, чтобы число нетерминалов грамматики в ослабленной нормальной форме Хомского было не очень велико, поскольку для каждого из них требуется создать булеву матрицу смежности. Например, в грамматике для анализа псевдонимов, переведённой в ослабленную нормальную форму Хомского, 12 нетерминалов. Однако число продукций в грамматике для Points-to анализа, учитывающего поля, зависит от числа полей анализируемой программы и может достигать нескольких тысяч. Это делает матричный алгоритм плохо применимым.

В алгоритме, основанном на произведении Кронекера, для пересечения рекурсивного автомата и ориентированного графа они представляются как композиция булевых матриц смежности для всех различных меток на рёбрах. На рис. 3 представлен рекурсивный автомат для Points-to анализа, учитывающего поля. Можно заметить, что все метки на рёбрах, кроме Alias, встречаются по одному разу. Поэтому для всех этих меток будет создана булева матрица смежности, содержащая единственный элемент, и для каждой из них необходимо считать произведение Кронекера с соответствующей матрицей смежности графа, а затем складывать полученный результат.

С целью уменьшения числа операций был рассмотрен другой способ представления графа и рекурсивного автомата в виде матриц. Стоит отметить, что в этом рекурсивном автомате переход между любыми двумя состояниями происходит только по одной метке. Это позволяет представить его одной матрицей смежности, элементы которой — целые числа. Старшие два бита этого числа содержат номер нетерминала, остальные — номер терминала (рис. 4).

В результате данного анализа между двумя вершинами в графе может быть добавлено ребро только с одним из трёх нетерминалов:

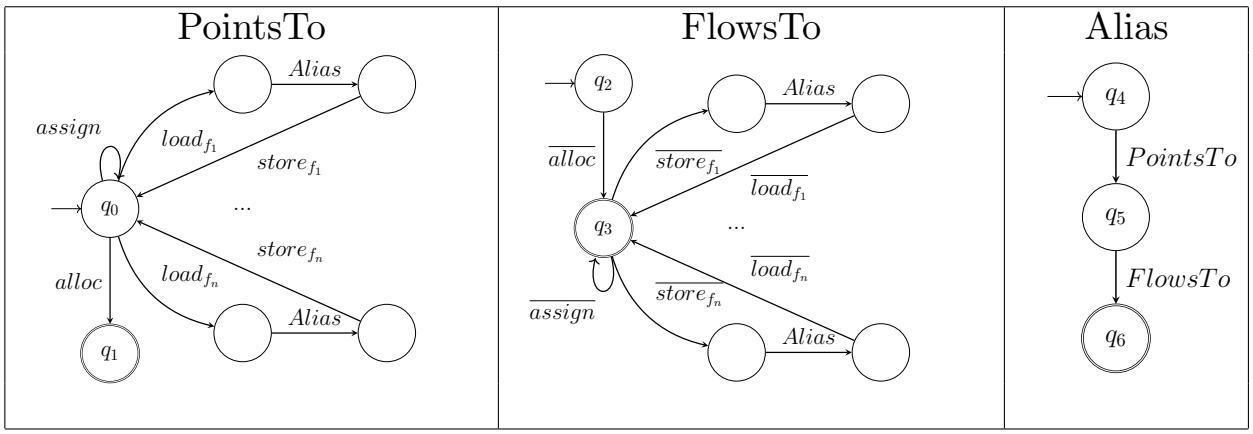


Рис. 3: Рекурсивный автомат для Points-to анализа, учитывающего поля

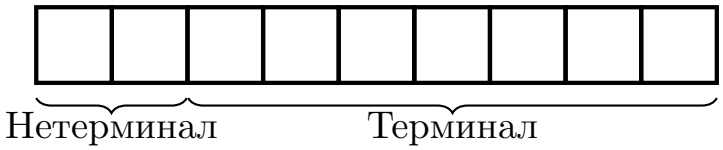


Рис. 4: Представление элементов матрицы смежности графа и рекурсивного автомата

- PointsTo — между вершиной, соответствующей переменной, и вершиной, соответствующей объекту кучи;
- FlowsTo — между вершиной, соответствующей объекту кучи, и вершиной, соответствующей переменной;
- Alias — между двумя вершинами, соответствующими переменным.

Если во входном графе между двумя вершинами будет не больше одного ребра, то для графа можно будет использовать такое же представление, как и для рекурсивного автомата. Однако способ построения графа по программе этого не гарантирует. Можно преобразовать входной граф так, чтобы он соответствовал этому требованию и по результату анализа этого графа можно было легко получить результаты для исходного. Если между парой вершин есть несколько рёбер, то можно разбить одно из них на два, добавив вершину, задающую новую переменную, как показано на рис. 5. Такое преобразование графа не влияет на результат анализа для исходных вершин.

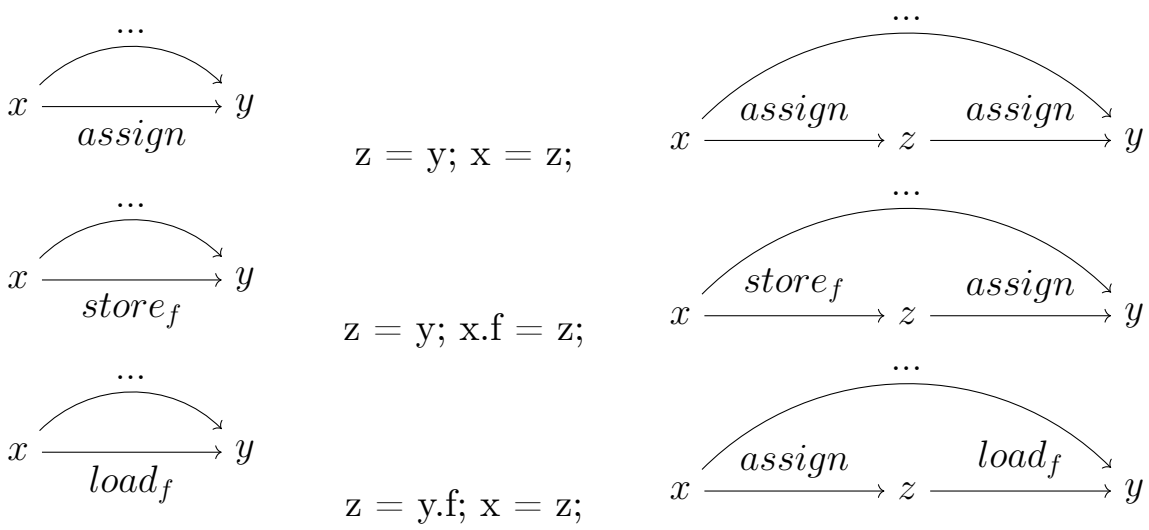


Рис. 5: Адаптация графа

Поскольку для пересечения автомата и графа применяется произведение Кронекера, необходимо задать бинарную операцию умножения для элементов матриц смежности описанного формата. Если ребро в графе и ребро в автомате содержат одинаковый нетерминал или одинаковый терминал, то соответствующая ячейка в результате произведения Кронекера будет $true$, иначе — $false$. Эту операцию можно задать так: $times(x, y) = (x_{nonterm} = y_{nonterm} \neq 0 \ or \ x_{term} = y_{term} \neq 0)$. Данное представление графа и рекурсивного автомата в виде матриц с применением такой операции позволяет на каждой итерации алгоритма вычислять произведение Кронекера всего один раз.

3.1. Применение модификации для анализа псевдонимов

На рис. 6 представлен рекурсивный автомат для анализа псевдонимов. В нём переход между любыми двумя состояниями происходит только по одной метке, поэтому для него можно использовать описанное выше представление. Однако в результате анализа между двумя вершинами могут быть добавлены как ребро MA, так и ребро VA. Поэтому необходимо иметь возможность хранить два нетерминала на ребре.

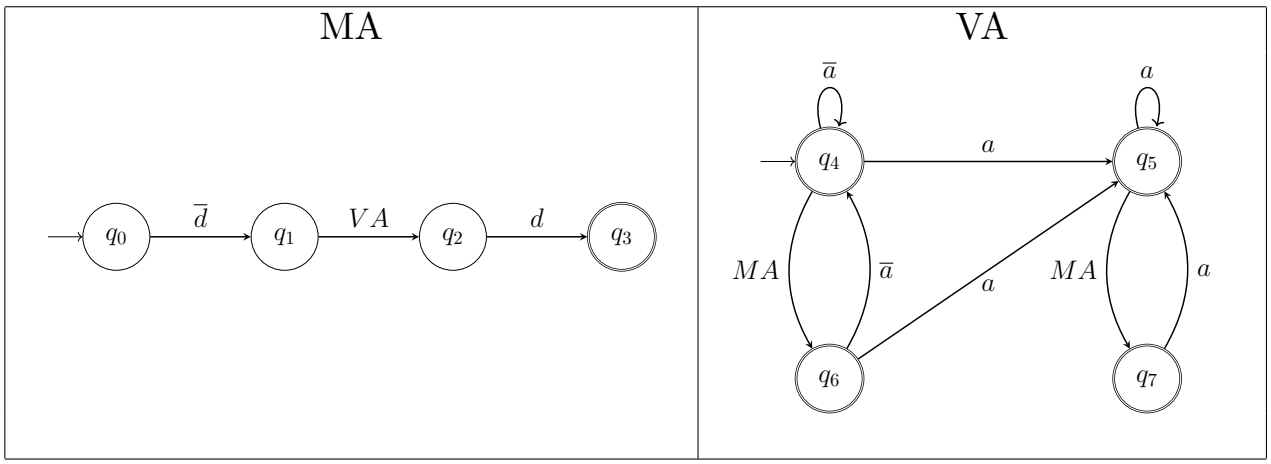


Рис. 6: Рекурсивный автомат для анализа псевдонимов

Чтобы хранить несколько нетерминалов на ребре, старшие биты числа, содержащего информацию о ребре, будут содержать не номер нетерминала, а маску для всех возможных нетерминалов. Тогда можно задать операцию умножения элементов для произведения Кронекера так: $times(x, y) = (x_{nonterm} \& y_{nonterm} \neq 0 \ or \ x_{term} = y_{term} \neq 0)$.

Стоит отметить, что такое представление также подойдёт для Points-to анализа, учитывающего поля, но оно потребует отдать под нетерминалы на один бит больше. Это может негативно сказаться на потреблении памяти: если под хранение терминала в числе не будет хватать памяти, придётся брать более длинный целочисленный тип для хранения.

3.2. Особенности реализации

Данная модификация тензорного алгоритма и её инкрементальная версия были реализованы в библиотеке CFPQ_PyAlgo[8], в которой для работы с разреженными матрицами используется библиотека pygraphblas, являющаяся python-обёрткой над библиотекой SuiteSparse.

При использовании предложенной операции умножения элементов матрицы после произведения Кронекера большинство элементов матрицы-результата будут $false$. Это приводит к двум проблемам:

[8] https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_PyAlgo/tree/vkutuev/ot_tensor, пользователь vkutuev. Дата посещения 22.05.2023

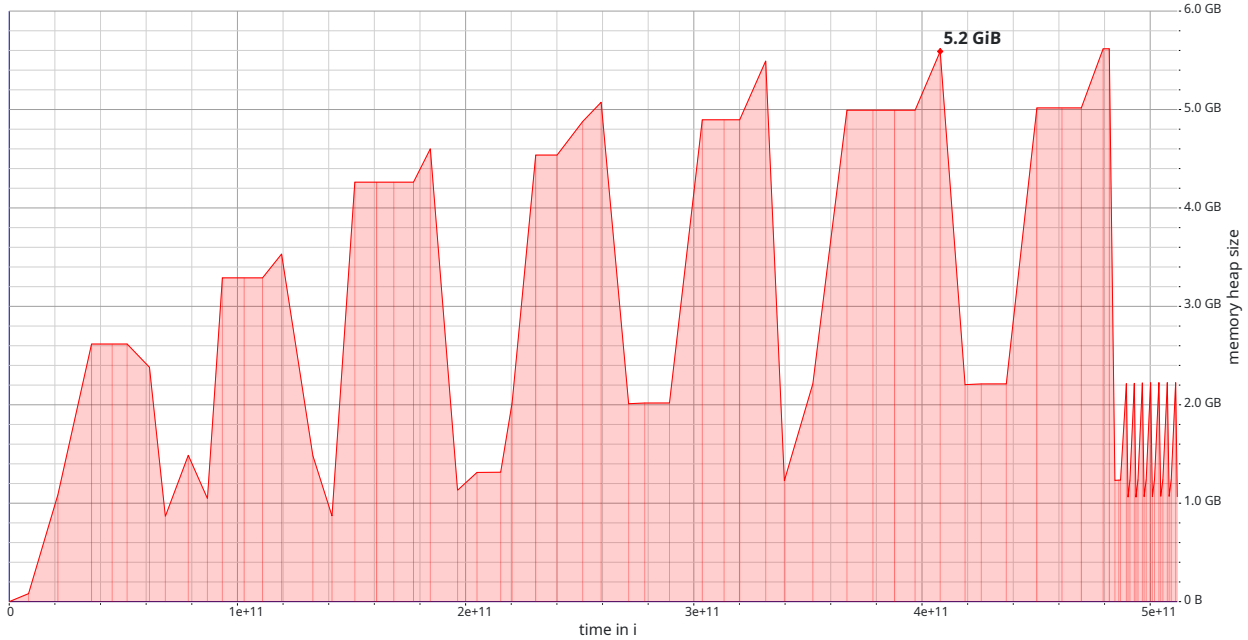
1. вычисление транзитивного замыкания потребует больше времени, так как матрица содержит больше элементов;

2. необходимо выделить больше памяти под хранение этой матрицы.

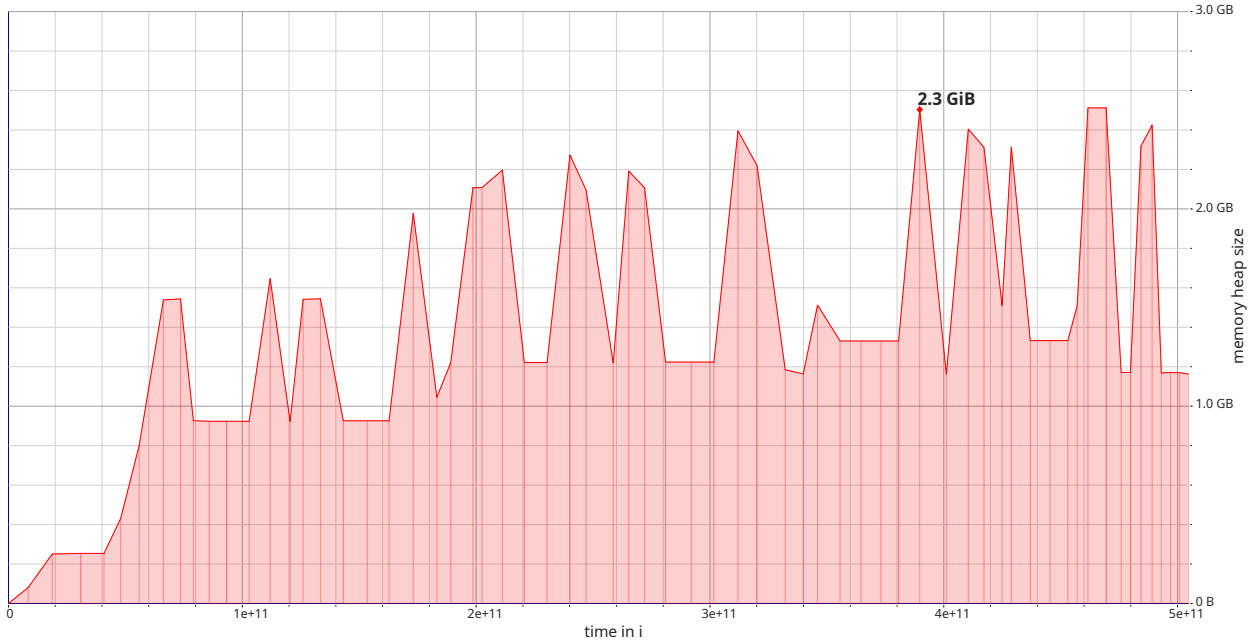
Очистка матрицы от значений $false$ после транзитивного замыкания поможет решить первую проблему. Однако потребность выделить дополнительную память под незначащие элементы матрицы может привести к нехватке памяти.

Для решения этой проблемы были внесены изменения в исходный код функции, вычисляющей произведение Кронекера, в библиотеке SuiteSparse[9]: перед выделением памяти под матрицу-результат производится вычисление количества значащих элементов в результате, и выделение памяти происходит только под это число элементов. Благодаря этому подходу удалось снизить пиковое потребление памяти. На рис. 7a представлен профиль памяти работы алгоритма со стандартной реализацией произведения Кронекера из библиотеки SuiteSparse с последующей фильтрацией результата, пиковое потребление памяти — 5,2 ГиБ. А на рис. 7b — профиль памяти работы алгоритма с доработанной реализацией, выделяющей память только под элементы результата, которые будут $true$, пиковое потребление памяти — 2,3 ГиБ.

[9] https://github.com/vkutuev/GraphBLAS/tree/vkutuev/kron, пользователь vkutuev. Дата посещения 22.05.2023



(a) Стандартная реализация



(b) Оптимизированная реализация

Рис. 7: Профиль потребления памяти на одном графе с разными реализациями произведения Кронекера

4. Оптимизация реализации матричного алгоритма

В pygraphblas при умножении двух матриц необходимо передать методу умножения полукольцо, в котором происходит сложение и умножение элементов матрицы. В реализации алгоритмов поиска путей с контекстно-свободными ограничениями используются разреженные булевы матрицы, поэтому для умножения использовалось полукольцо BOOL.LOR_LAND: домен — булевы значения $\{true, false\}$, сложение — LOR (дизъюнкция), умножение — LAND (конъюнкция). Однако разреженность матриц меняет множество значений, которые может принимать элемент матрицы: $true$, $false$ и «нет значения». При этом в реализации матричного алгоритма элементы матрицы никогда не принимают значения $false$. Изначально во всех матрицах нет значений, затем некоторые элементы матрицы в соответствии с простыми правилами грамматики инициализируются значением $true$. Затем в матрицу могут добавляться только значения $true$, если была найдена новая пара вершин, между которыми существует путь. Таким образом, полукольцо BOOL.LOR_LAND является избыточным для реализации данного алгоритма. Для избавления от этой избыточности можно воспользоваться полукольцом с операциями ANY (выбирает любой из переданных аргументов) в качестве сложения и PAIR (возвращает 1 в полукольце, если оба операнда — присутствующие в матрице элементы) в качестве умножения. В силу того, как заполняется матрица в данном алгоритме, использование полукольца BOOL.ANY_PAIR при умножении матриц не изменит результат умножения, но операции ANY и PAIR проще и выполняются быстрее, чем LOR и LAND.

Данная оптимизация была применена в реализации матричного алгоритма в CFPQ_PyAlgo[10].

[10] https://github.com/FormalLanguageConstrainedPathQuerying/CFPQ_PyAlgo, пользователь vkutuev. Дата посещения 22.05.2023

5. Эксперименты

5.1. Тестовый стенд

Для замеров времени работы алгоритмов использовался компьютер со следующими характеристиками:

- CPU: Intel(R) Core(TM) i7-479, тактовая частота: 3.6 ГГц, 4 ядра, 4 потока;
- ОЗУ: 64 ГиБ;
- GPU: GeForce GTX 1070 GPU (8 ГиБ памяти DDR5).

Для проведения экспериментов использовалось следующее программное обеспечение:

- операционная система: Ubuntu 20.04;
- Python: 3.8.10;
- Java: 15.0.2;
- GCC: 9.4.0.

5.2. Тестовые данные

Для проведения экспериментов были взяты графы из набора CFPQ_Data, построенные по реальным программам. Описание графов для анализа псевдонимов и Points-to анализа, учитывающего поля, представлены в таблицах 1 и 2.

5.3. Замеры времени работы и потреблетния памяти

Для проведения замеров производительности каждая реализация запускалась на каждом анализируемом графе по 30 раз. При замерах времени работы учитывалось только время обработки уже загруженных в память графа и грамматики.

Граф	$\|V\|$	$\|E\|$	Число пар достижимых вершин
wc	332	269	156
bzip2	632	556	315
pr	815	692	385
ls	1 687	1 453	854
gzip	2 687	2 293	1 458
apache	1 721 418	1 510 411	92 806 768
init	2 446 224	2 112 809	3 783 769
mm	2 538 243	2 191 079	3 990 305
ipc	3 401 022	2 931 498	5 249 389
lib	3 401 355	2 931 880	5 276 303
block	3 423 234	2 951 393	5 351 409
arch	3 448 422	2 970 242	5 339 563
crypto	3 464 970	2 988 387	5 428 237
security	3 479 982	3 003 326	5 593 387
sound	3 528 861	3 049 732	6 085 269
net	4 039 470	3 500 141	8 833 403
fs	4 177 416	3 609 373	9 646 475
drivers	4 273 803	3 707 769	18 825 025
postgre	5 203 419	4 678 543	90 661 446
kernel	11 254 434	9 484 213	16 747 731

Таблица 1: Описание графов из тестового набора для анализа псевдонимов

Граф	$\|V\|$	$\|E\|$	Число пар достижимых вершин
sunflow	15464	15957	16354
lusearch	15774	14994	9242
luindex	18532	17375	9677
avrora	24690	25196	21532
eclipse	41383	40200	21830
batik	60175	63089	45968
fop	86183	83016	76615
pmd	54444	59329	60518
xalan	58476	62758	52382
h2	44717	56683	92038
tomcat	111327	110884	82424

Таблица 2: Описание графов из тестового набора для Points-to анализа, учитывающего поля

Для замеров времени работы реализаций из CFPQ_PyAlgo использовался пакет benchmark, входящий в репозиторий, позволяющий задать замеряемую реализацию, входные данные и количество измерений, которые необходимо провести. Для замеров памяти CPU-реализаций измерялся Resident Set Size (RSS) процесса, а для GPU-реализаций измерялось потребление видеопамяти профилятором[11], основанным на nvidia-smi[12].

Для проведения замеров времени работы и потребляемой памяти реализации, основанной на алгоритме синтаксического анализа GLL, в реализацию были внесены изменения[13], позволяющие проводить замеры времени работы и потребления памяти с произвольной контекстно-свободной грамматикой, с указанным хранилищем для графа, с заданным числом итераций для прогрева JVM и для замеров.

Graspan позволяет получить время обработки загруженного в память графа, однако не позволяет получить информацию о потребляемой памяти. Для получений объёма затраченной памяти измерялся RSS процесса.

Gigascale позволяет получить как время обработки загруженного графа, так и объём затраченной памяти.

5.4. Оптимизация реализации матричного алгоритма

Чтобы оценить влияние предложенной оптимизации, было измерено время работы с ней и без неё. Для сравнения реализаций были взяты графы apache и init, а также графы меньшего размера из CFPQ_Data. Результаты замеров с использованием разных полуколец для перемножения матриц представлены в таблице 3.

Полученные результаты показывают, что для графов небольшого

[11] https://github.com/EgorOrachyov/SpBench/blob/main/scripts/gpu_mem_prof.py, дата посещения 22.05.2023

[12] https://developer.nvidia.com/nvidia-system-management-interface, дата посещения 22.05.2023

[13] https://github.com/FormalLanguageConstrainedPathQuerying/GLL4Graph/tree/vkutuev/memory_usage, пользователь vkutuev. Дата посещения 22.05.2023

Граф	LOR_LAND (сек.)	ANY_PAIR (сек.)	Ускорение
wc	0,006	0,006	1,00
bzip2	0,022	0,022	1,00
pr	0,013	0,012	1,08
ls	0,051	0,045	1,13
gzip	0,038	0,030	1,26
apache	683,58	536,70	1,27
init	59,33	45,84	1,29

Таблица 3: Влияние оптимизации на время работы

размера данная оптимизация практически не влияет на время работы, однако с ростом графов возрастает и ускорение, которое она даёт.

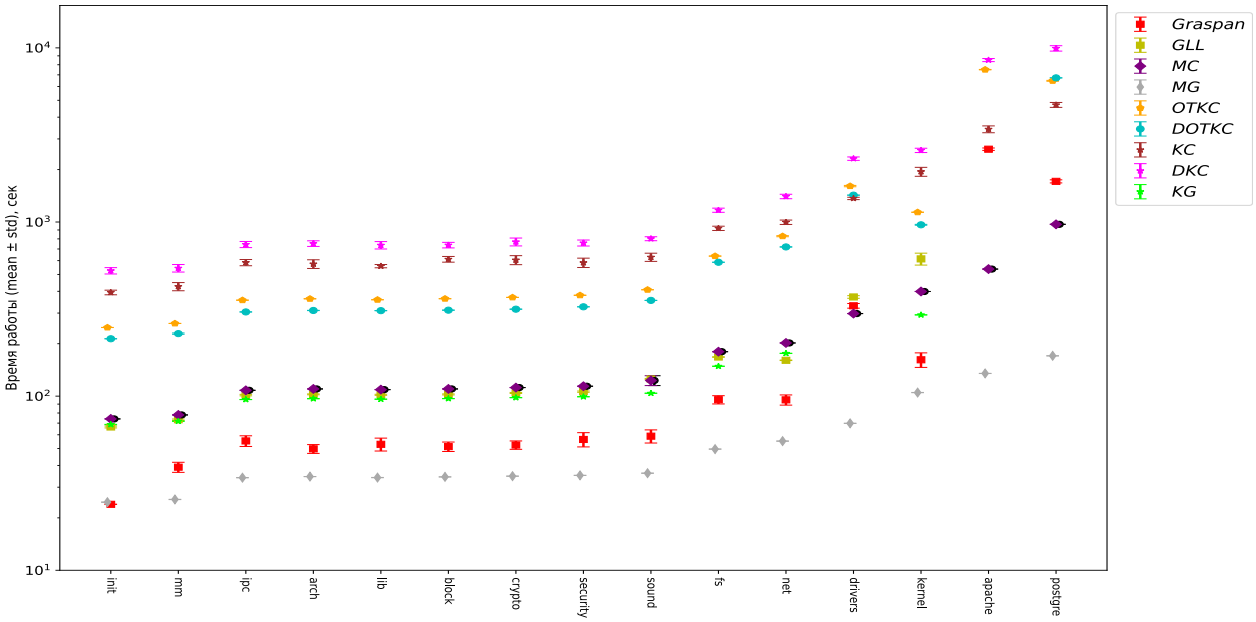
5.5. Сравнение алгоритмов KC-достижимости

В ходе экспериментов сравнивались следующие реализации:

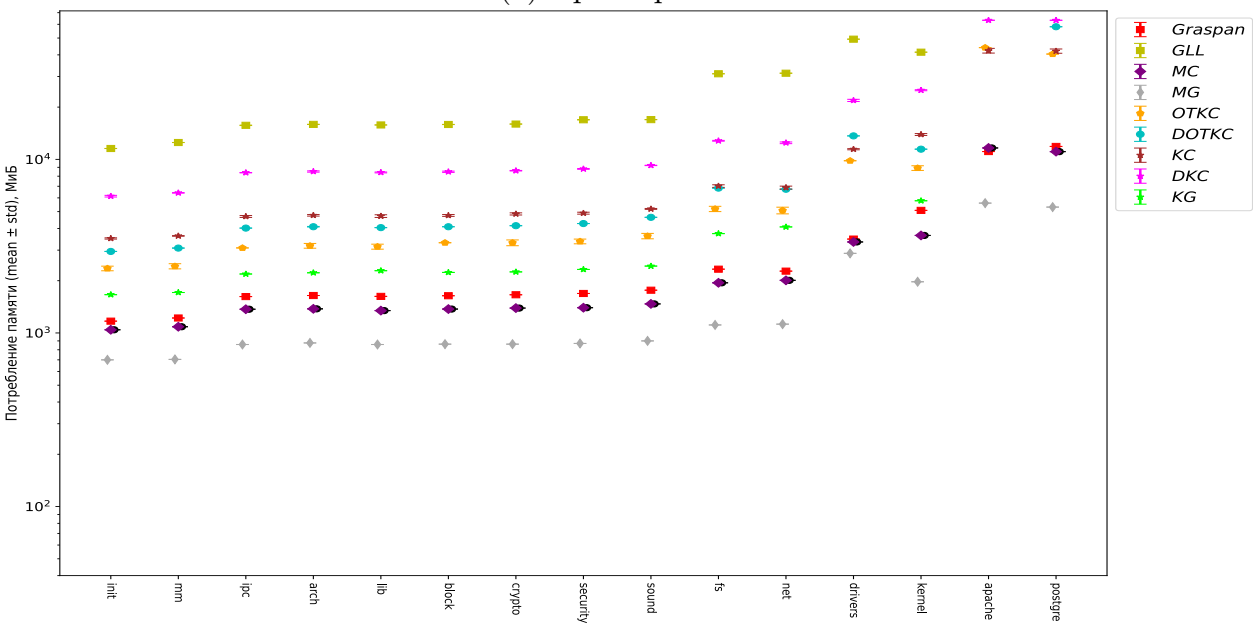
- реализации матричного алгоритма из CFPQ_PyAlgo для CPU (**MC**) и GPU (**MG**);
- реализация тензорного алгоритма из CFPQ_PyAlgo для CPU (**KC**) и GPU (**KG**), инкрементальная версия тензорного алгоритма (**DKC**);
- реализация адаптированного для Points-to анализа, учитывающего поля, тензорного алгоритма (**OTKC**) и его инкрементальная версия (**OTDKC**);
- реализация алгоритма, основанного на алгоритме синтаксического анализа **GLL** (запускалась вариация с хранением графа в оперативной памяти);
- **Graspan** (запускался только для анализа псевдонимов, так как эта реализация не поддерживает грамматики с большим количеством нетерминалов);

- **Gigascale** (запускался только для Points-to анализа, учитывающего поля, так как эта реализация заточена под конкретную грамматику).

На рис. 8 и 9 представлены результаты замеров времени работы и потребления памяти данных реализаций. Результаты некоторых запусков не отображаются, если время работы превышает 5 часов или запуск завершился с ошибкой из-за нехватки памяти.

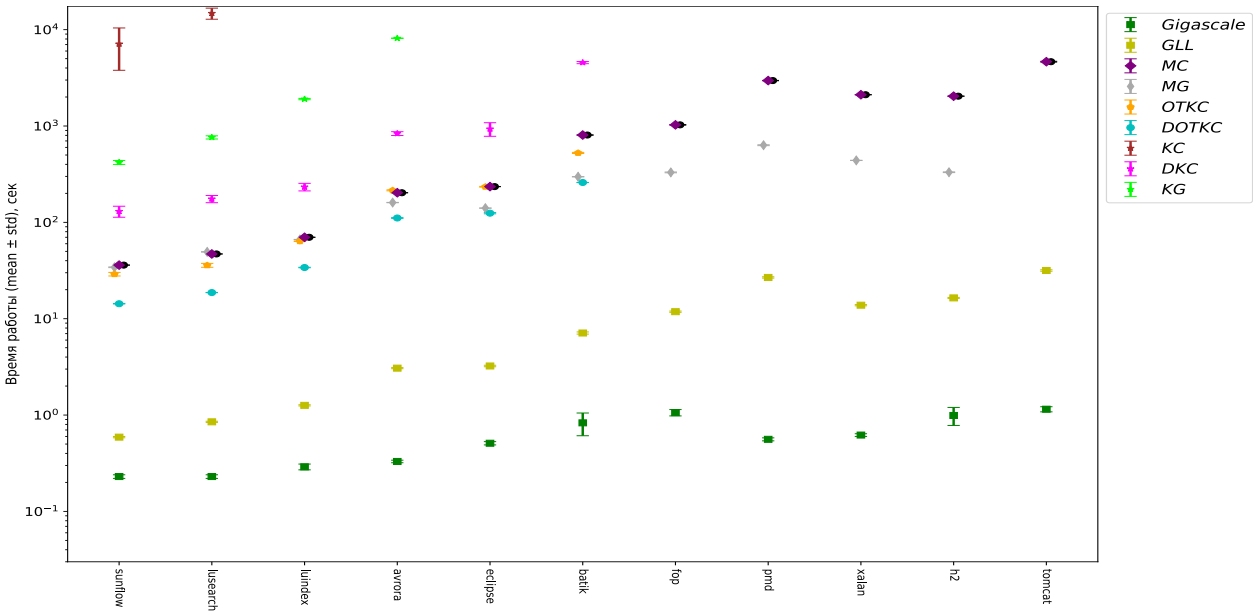


(a) Время работы

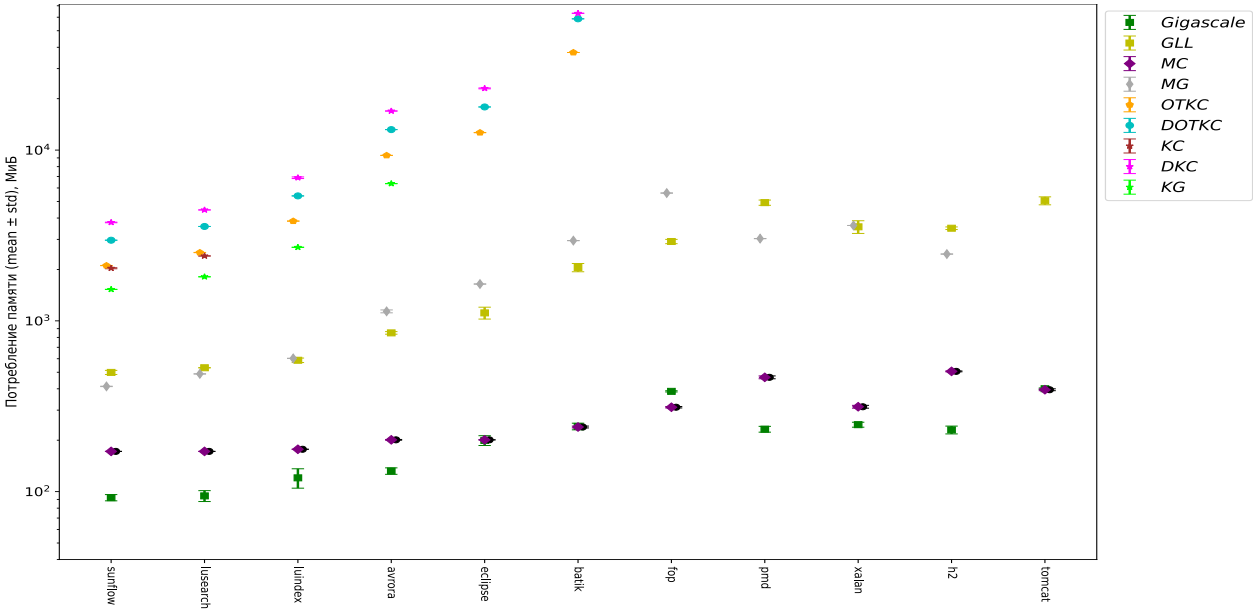


(b) Потребление памяти

Рис. 8: Результаты для анализа псевдонимов



(a) Время работы



(b) Потребление памяти

Рис. 9: Результаты для Points-to анализа, учитывающего поля

На основе полученных в результате экспериментов данных можно сделать следующие выводы.

- При анализе псевдонимов наилучшие результаты (как время работы, так и потребление памяти) показала реализация матричного алгоритма для GPU. Для CPU же наименьшее время работы продемонстрировал Graspan. Однако при наличии в ответе большого числа пар достижимых вершин он работает значительно медлен-

нее реализаций матричного алгоритма.

- Наименьшие время работы и потребление памяти при Points-to анализе, учитывающем поля, показала реализация Gigascale, работающая значительно быстрее всех остальных реализаций, что вполне обоснованно, ведь данный инструмент был разработан именно для такого анализа.

- Предложенные в работе модификация тензорного алгоритма и её инкрементальная версия работают быстрее других реализаций алгоритма для CPU в обоих видах анализа. Также можно отметить, что на небольших графах для Points-to анализа, учитывающего поля, инкрементальная версия продемонстрировала наименьшее время работы среди всех алгоритмов, основанных на операциях линейной алгебры. Однако эта реализация, как и базовый тензорный алгоритм, обладает таким недостатком, как большое потребление памяти. Это связано с большим размером получаемой в результате произведения Кронекера матрицы, который зависит и от размеров графа, и от размеров грамматики, что привело к ошибке нехватки памяти на графах fop, pmd, xalan, h2 и tomcat.

Заключение

В ходе работы были достигнуты следующие результаты.

- Предложена модификация тензорного алгоритма для Points-to анализа, учитывающего поля, показавшая наилучшее время работы на небольших графах среди алгоритмов, основанных на операциях линейной алгебры. Однако эксперименты показали, что из-за большого потребления памяти алгоритм практически неприменим для данного вида анализа.

- Оптимизирована реализация матричного алгоритма из библиотеки CFPQ_PyAlgo, эффективность оптимизации экспериментально проверена.

- Проведены замеры производительности реализаций алгоритмов КС-достижимости, которые показали, что алгоритмы, основанные на операциях линейной алгебры, достаточно эффективны для анализа псевдонимов, но из-за больших размеров грамматики малоэффективны для Points-to анализа, учитывающего поля.

Так как реализации алгоритма, основанного на произведении Кронекера, и его вариации плохо применимы для Points-to анализа, учитывающего поля, в силу очень большого объёма потребляемой памяти под пересечение графа и рекурсивного автомата, в дальнейшем можно рассмотреть следующие шаги по оптимизации матричного алгоритма для применения его к данному типу анализа.

- Разработка инкрементальной версии матричного алгоритма, в которой полученные на предыдущих итерациях данные не будут пересчитываться в дальнейшем, что позволит сократить затрачиваемое на умножение матриц время.

- Специализация матричного алгоритма для данного анализа, не требующего перевода грамматики в ОНФХ и вычисления достижимости для нетерминала $FlowsTo$ непосредственно, что позволит значительно сократить количество перемножений матриц.

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] 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 (online; accessed: 10.10.2021).

[3] 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.

[4] 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 : Association for Computing Machinery, 2019. — 5 p. — URL: https://doi.org/10.1145/3327964.3328503 (online; accessed: 10.10.2021).

[5] Faults in Linux: Ten Years Later / Nicolas Palix, Gaël Thomas, Suman Saha et al. // Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems. — ASPLOS XVI. — New York, NY, USA : Association for Computing Machinery, 2011. — P. 305–318. — URL: https://doi.org/10.1145/1950365.1950401 (online; accessed: 10.10.2021).

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

[7] Hopcroft John E., Motwani Rajeev, Ullman Jeffrey D. Introduction to Automata Theory, Languages, and Computation (3rd Edition). — USA : Addison-Wesley Longman Publishing Co., Inc., 2006. — ISBN: 0321455363.

[8] Medeiros Ciro M., Musicante Martin A., Costa Umberto S. An Algorithm for Context-Free Path Queries over Graph Databases // Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity. — SBLP '20. — New York, NY, USA : Association for Computing Machinery, 2020. — P. 40–47. — URL: https://doi.org/10.1145/3427081.3427087 (online; accessed: 10.10.2021).

[9] 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. — 2103.14688.

[10] Reps Thomas. Program analysis via graph reachability1An abbreviated version of this paper appeared as an invited paper in the Proceedings of the 1997 International Symposium on Logic Programming [84].1 // Information and Software Technology. — 1998. — Vol. 40, no. 11. — P. 701–726. — URL: https://www.sciencedirect.com/science/article/pii/S0950584998000937 (online; accessed: 10.10.2021).

[11] Reps Thomas, Horwitz Susan, Sagiv Mooly. Precise Interprocedural Dataflow Analysis via Graph Reachability // Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '95. — New York, NY, USA : Association for Computing Machinery, 1995. — P. 49–61. — URL: https://doi.org/10.1145/199448.199462 (online; accessed: 10.10.2021).

[12] Sagiv Mooly, Reps Thomas, Horwitz Susan. Precise interprocedural dataflow analysis with applications to constant propagation // Theoretical Computer Science. — 1996. — Vol. 167, no. 1. — P. 131–170. — URL: https://www.sciencedirect.com/science/article/pii/0304397596000722 (online; accessed: 10.10.2021).

[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 (online; accessed: 10.10.2021).

[14] Sridharan Manu, Bodík Rastislav. Refinement-Based Context-Sensitive Points-to Analysis for Java // Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '06. — New York, NY, USA : Association for Computing Machinery, 2006. — P. 387–400. — URL: https://doi.org/10.1145/1133981.1134027 (online; accessed: 10.10.2021).

[15] Zheng Xin, Rugina Radu. Demand-Driven Alias Analysis for C // SIGPLAN Not. — 2008. — 1. — Vol. 43, no. 1. — P. 197–208. — URL: https://doi.org/10.1145/1328897.1328464 (online; accessed: 10.10.2021).