Оглавление

Введение

Актуальность

Графовые модели данных широко используются в различных областях науки, например, в биоинформатике [45], анализе социальных сетей [57, 11], графовых базах данных [31, 51] и разных видах статического анализа [39].

Одной из важных задач в анализе графовых моделей данных является поиск путей с заданными ограничениями. Одним из способов задавать такие ограничения являются формальные языки: если на рёбрах графа написаны метки из фиксированного алфавита, то можно искать пути, конкатенация меток на которых принадлежит фиксированному языку [7]. Например, хорошо изучена задача поиска путей с ограничениями, заданными регулярными языками [33]. В этой же работе мы остановимся на классе контекстно-свободных языков, так как они позволяют решать более широкий класс задач.

Задача поиска путей с контекстно-свободными ограничениями, или, сокращённо, CFPQ[1] была впервые сформулирована в терминах запросов к графовым базам данных [51], но нашла применение и в прочих отраслях, использующих графовые модели. За более чем 30 лет, прошедших с тех пор, было предложено множество разных алгоритмов для её решения [32, 21, 42], в большинстве своём основанных на различных методах синтаксического анализа.

К сожалению, все существующие решения задачи в общем случае недостаточно эффективны для использования на практике [18]. Более того, существует условная нижная оценка [20], согласно которой, скорее всего, достаточно быстрых решений задачи CFPQ в общем случае и не существует.

Всё вышесказанное приводит к тому, что имеет смысл разрабатывать алгоритмы для частных случаев задачи, имеющие время работы лучше, чем общее решение. Для некоторых из этих случаев уже были построены подобные решения, например для языков Дика на двунаправленных графах [55, 10]. Проблема существующих решений в том, что они слишком Ad hoc, то есть построены специально под конкретный частный случай, а потому применённые в них подходы и идеи невозможно переиспользовать при построении решений для других частных случаев.

Данная работа нацелена на создание единого подхода к построению решений задачи для CFPQ и применение этого подхода для разработки на его основе новых решений для некоторых частных случаев задачи.

[1] Context-Free Path Querying

Постановка задачи

Определение 0.1. *Ориентированный помеченный граф* (или *граф с метками*) — это тройка $G = \langle V, E, \Sigma \rangle$, где V — множество вершин, Σ — множество меток, $E \subseteq V \times V \times \Sigma$ — множество рёбер.

Неформально, это мультиграф, каждому ребру которого сопоставлена метка из алфавита Σ.

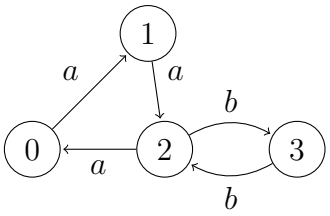


Рис. 1: Помеченный граф с $\Sigma = \{a, b\}$

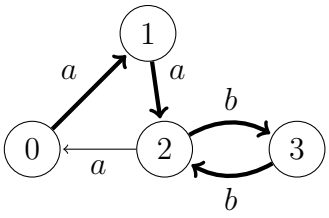


Рис. 2: На пути $v_0 \to v_1 \to v_2 \to v_3 \to v_2$ читается слово $aabb$

Определение 0.2. Будем говорить, что слово w *читается* на пути p, если конкатенация меток вдоль p образует w.

Определение 0.3. *Контекстно-свободная грамматика* — это четвёрка $\langle \Sigma, N, S, P \rangle$:

- Σ — конечный алфавит
- N — конечное множество нетерминалов
- $S \in N$ — стартовый нетерминал
- P — конечное множество продукций (правил вывода), имеющих вид $N_i \to \alpha$, где $N_i \in N, \alpha \in (\Sigma \cup N)^*$

Язык, распознаваемый грамматикой $\mathcal{G}$ — язык $L(G)$ слов, выводимых из стартового нетерминала.

Контекстно-свободная грамматика задаёт *контекстно-свободный язык*.

TODO: пример

TODO: в примере пояснить, что такое ε

Теперь определим саму задачу.

Определение 0.4. Дан ориентированный помеченный граф G и контекстно-свободная грамматика $\mathcal{G}$ над тем же алфавитом.

Задача поиска путей с контекстно-свободными ограничениями (CFPQ)[1] заключается в нахождении всех пар вершин $u, v \in V(G)$, таких что существует путь $p : u \rightsquigarrow v$, на котором читается слово $w \in L(\mathcal{G})$.

TODO: пример

[1] В реляционной семантике [22]

Цель и задачи

Целью работы является получить решения для частных случаев в задаче поиска путей с контекстно-свободными ограничениями, основываясь на едином подходе.

Для её достижения решаются следующие задачи:

- Выбрать единый подход для решения задачи CFPQ
- Построить на его основе решения для следующих частных случаев:
  - Язык Дика $\mathcal{D}_k$ и двунаправленные графы
  - Язык Дика $\mathcal{D}_1$ (на одном типе скобок)
- Применить выбранного подхода для решение задачи достижимости для смешанный языка Дика $\mathcal{D}_1 \odot \mathcal{D}_1$ на двунаправленных графах

TODO: Всё тут переписать

Структура работы

В главе 1 определены ключевые термины и дана формальная постановка задачи.

В главе 2 приведена история развития области и анализ существующих решений для задачи CFPQ, а также обоснован выбор подхода к построению новых частичных решений.

В главе 3 представлен выбранный подход к решению задачи CFPQ (идея пересечения языков).

TODO

1. Обзор литературы

Задача поиска путей с контекстно-свободными ограничениями (она же CFPQ) была сформулирована Михалисом Яннакакисом ещё в 1990 году [51] в применении к запросам к декларативному языку Datalog [48, 9], textitно эта идея не была особо развита.

В последнее десятилетие интерес к задаче *вспыхнул с новой силой* в контексте запросов к RDF[1] [37] — графовой модели представления данных в сети, разработанной W3C[2]. RDF хранит объекты (ресурсы) и утверждения об их связях (тройки «субъект — предикат — объект»). Чаще всего для работы с данными, представленными в RDF, используют язык запросов SPARQL [41]. Однако SPARQL позволяет осуществлять только запросы, представленные в виде регулярных выражений, тогда как некоторые интересные запросы (такие как *same-generation queries*[3] [1]) могут быть выражены только в терминах контекстно-свободных языков.

Задача также нашла широкое применение в разных видах статического анализа [39] (в это области она более известная под именем задачи контекстно-свободной достижимости или CFL-Reachability), таких как

TODO: написать про виды стат анализа..

1.1. Решения задачи в общем случае

За более чем 30 лет было предложено множество алгоритмов для решения задачи CFPQ.

Большая часть решений реализует идеи различных алгоритмов разбора выражений (парсинга). Так, алгоритм Мельски и Репса [32] использует тот же подход, что и алгоритм Кока-Янгера-Касами [54] парсинга КС-языков: приведение грамматики к нормальной форме Хомского [15] и подсчёт динамического программирования — и имеет ту же асимптотику $\mathcal{O}(|V|^3|N|^3)$, где $|V|$ — число вершин в графе, $|N|$ — число нетерминалов входной грамматики. Позднее Чаудхури [12] улучшил этот алгоритм, уменьшив асимптотику в $\log|V|$ раз, используя метод четырёх русских [35].

Алгоритм Григорьева и Рогозиной, основан на обобщённом нисходящем синтаксическом анализе[4] — GLL [43] парсинге, и работает за $\mathcal{O}(|V|^3 \max v \in V(deg^+(v)))$. Алгоритм Медейроса и др. [31] также основан на нисходящем синтаксическом анализе — LL парсинге, но имеет время работы $\mathcal{O}(|V|^3|P|)$, где $|P|$ — число продукций входной грамматики.

Алгоритм Сантоса и др. [42] основан на восходящем синтаксическом анализе[5] —

[1] Resource Description Framework — Среда описания ресурса
[2] World Wide Web Consortium — Консорциум Всемирной паутины
[3] Запросы поиска объектов, находящихся на одном уровне иерархии
[4] Top-down parsing
[5] Bottom-up parsing

Tomita-Style Generalized LR парсинге [44], однако его время работы не оценено (**TODO**: точно?), а на некоторых входах он вообще не завершается (однако на практике показывает хорошую производительность).

Но есть и подходы, не основанные на алгоритмах парсинга. Например, в своей работе Азимов и Григорьев [42] сводят задачу CFPQ к транзитивному замыканию матриц (по аналогии с решением Валианта [47] задачи распознавания КС-языков). Преимущество этого алгоритма в том, что он использует операции над матрицами, которые могут быть соптимизированы с использованием GPGPU[1].

Хеллингс в своей работе [21] рассматривает задачу в основанной на путях (path-based) семантике запроса и разрешает её, используя аннотированные грамматики.

Чаудхури [13], а также Орачев и др. [16] сводят задачу CFPQ к задаче достижимости в рекурсивном конечном автомате, которую решают, используя инкрементальное транзитивное замыкание. Наивная реализация работает за $\mathcal{O}(|V|^3|N|^3)$, но так же (как и алгоритм Репса [32, 12]) может быть соптимизирован [36] в $\log|NV|$ раз методом четырёх русских.

1.2. Нижние оценки

Как можно заметить, все существующие алгоритмы (для решения CFPQ в общем случае) имеют кубическое (или большее) время работы. Что не достаточно эффективно для работы с реальными данными, как экспериментально показали Кёйперс и др. [18], реализовав и замерив производительность трёх алгоритмов: Хеллингса [21], Сантоса и др. [42] и Азимова и др. [42].

Более того, скорее всего, решения с более быстрой ($\mathcal{O}(|V|^{3-\varepsilon})$) асимптотикой не существует. Это так называемый "cubic bottleneck"[2] [20] данной задачи. Было доказано, что она является 2NPDA[3]-полной, и субкубическое решение для неё повлечёт наличие субкубических алгоритмов для всех задач класса. Учитывая, что такие решения не были найдены за более чем 50 лет, маловероятно, что данная задача решается быстрее куба.

Существуют и другие условные нижние оценки.

Так, Чаттерджи и др. в своей работе [10] построили условную нижнюю оценку, сведя в к задаче CFPQ (а именно, к $\mathcal{D}_k$-достижимости (опр. 1.2)) задачу BMM[4], на которую есть условная нижняя оценка. А именно, согласно BMM-гипотезе [49], не существует субкубического *комбинаторного*[5] алгоритма для перемножения двух булевых

[1] General-purpose computing on graphics processing units — техника использования графического процессора для неграфических целей (математических вычислений)

[2] узкое место

[3] 2-way nondeterministic pushdown automata [3] — 2-сторонний автомат с магазинной памятью

[4] Boolean Matrix Multiplication — перемножение двух булевых матриц

[5] Этот термин не вполне определен, но можно понимать его как "не алгебраический". В частности, комбинаторные алгоритмы не должны использовать деление и вычитание, так те пользуются особен-

матриц. Замечание про комбинаторность алгоритма важно, так как алгебраическое субкубическое решения для BMM существует, а именно, она сводится к обычному перемножению матриц, которое может быть совершено за $\mathcal{O}(|V|^\omega)$[1].

Позднее Чжан [58] улучшил эту оценку, построив сведение BMM к $\mathcal{D}_1$-достижимости, тем показав, что, скорее всего, не существует субкубического комбинаторного решения уже для неё.

И если с комбинаторными алгоритмами всё более менее понятно и оценки (нижняя и верхняя) сходятся, то с некомбинаторными всё не так радужно. Нижняя оценка в $\mathcal{O}(|V|^\omega)$ вытекает из сведения от BMM, но обратного сведения не построено и непонятно, может ли эта оценка быть достигнута. Более того, существует условная нижняя оценка на нижнюю оценку: Чистиков и др. [14] показали, что при условии NSETH[2] [34] не существует нижней оценки, основанной на SETH[3] [26] для $\mathcal{D}_2$-достижимости, лучшей, чем $\mathcal{O}(|V|^\omega)$.

1.3. Решения задачи в частных случаях

Понятно, что для решения практических задач далеко не всегда нужна CFPQ в общем случае. Чаще всего для каждой конкретной задачи нужна конкретная КС грамматика, а иногда ещё и понятны ограничения на тип графа.

Пользуясь этой информацией (ограничениями на тип грамматики и графа) можно конструировать частные, более быстрые решения, или давать более точные оценки на время работы существующих.

Начнём с очевидных частных случаев. Так, для графа-цепочки задача CFPQ — это просто задача КС-распознавания, так как граф-цепочка — это просто одна строка. А для задачи КС-распознавания существует субкубическое решение [47], более того, она эквивалентна задаче перемножения булевых матриц (так что решается за $\mathcal{O}(|V|^\omega)$). В своей работе [51] Яннакакис также заметил, что это сведение можно обобщить на случай ациклических графов.

Решение с такой же асимптотикой $\mathcal{O}(|V|^\omega)$ получено для ещё одного частного случая — иерархических грамматик[4] [52]. На самом деле такие грамматики задают регулярные языки, однако размер автомата может быть экпоненциален относительно размера грамматики. Но если считать размер грамматики константным (как делают довольно часто), то задача поиска путей с регулярными ограничениями решается за $\mathcal{O}(V^\omega)$, так как сводится [51] к построению транзитивного замыкания входного графа (опр. **TODO**).

ностями алгебраических структур (а именно, существованием обратного)

[1] $\omega < 2.373$ [4]

[2] Nondeterministic Strong Exponential Time Hypothesis — Недетерминированная сильная гипотеза об экспоненциальном времени

[3] Strong exponential time hypothesis — Сильная гипотеза об экспоненциальном времени

[4] Hierarchical state machines

Ещё одним из языков, для которых строятся частичные решения, является язык Дика.

Определение 1.1. *Языком Дика*[1] на k типах скобок $\mathcal{D}_k$ называют язык, заданный над алфавитом $\Sigma_k = \{(_1,)_1, (_2,)_2 \dots (_k,)_k\}$ и состоящий из правильных скобочных последовательностей на k типах скобок.

Язык Дика — контекстно-свободный и задаётся следующей грамматикой:
$\mathcal{D}_k : S ::= SS \mid (_1 S)_1 \mid (_2 S)_2 \mid \dots (_k S)_k \mid \varepsilon$

Определение 1.2. Задачу CFPQ для языка Дика называют также задачей Диковой достижимости[2] [28] или $\mathcal{D}_k$-достижимости.

Задача Диковой достижимости широко применяется в статическом анализе, так как во многих видах анализа возникают парные объекты: вызовы и возвраты из функций [46], обращение по указателю и их разыменование [59], запись и чтение из поля [50], взятие и возврат блокировки [27].

Первыми улучшениями для задачи Диковой достижимости стали лучшие оценки на время работы уже существующих алгоритмов. Так, Кодумал и Айкен [28] показали, что алгоритм Репса [32], применённый для грамматики Дика (размер которой $\mathcal{O}(k)$ для языка $\mathcal{D}_k$), имеет асимптотику $\mathcal{O}(|V|^3 k)$, тогда как обычная оценка — $\mathcal{O}(|V|^3 k^3)$. А Рехоф и Фендрих [38] показали оценку (также для алгоритма Репса) в $\mathcal{O}(|V|^3)$ для языка Дика и графов потока исполнения, построенных для решения задачи основанного на типах анализа потока[3].

Как уже было сказано выше, существует кубическая нижняя оценка на время работы *комбинаторного* алгоритма уже для задачи Диковой достижимости. Это однако не мешает существованию субкубических алгебраических алгоритмов. Так, для задачи достижимости для языка Дика *на одном типе скобок* $\mathcal{D}_1$ были построены более быстрые алгоритмы. Бредфорд сводит [8] задачу $\mathcal{D}_1$-достижимости к $\mathcal{O}(\log^2 |V|)$ перемножениям AGMY-матриц[4], каждое из которых может быть произведено за $\mathcal{O}(|V|^\omega \log |V|)$. Матиасен и др. [30] строят "более комбинаторное" решения, строя сведение $\mathcal{D}_1$-достижимости к $\mathcal{O}(\log^2)$ непосредственно перемножений булевых матриц (для подсчёта транзитивного замыкания графа [2]).

В общем же случае (для языка Дика более, чем на одном типе скобок) субкубических решений не существует. Однако они появляются при добавлении ограничений на вид графа. А именно, при рассмотрении двунаправленных графов.

Определение 1.3. Помеченный граф $G = \langle V, E, \Sigma_k \rangle$ называют *двунаправленным*[5] [55],

[1] Dyck language
[2] Dyck-reachability
[3] Type-based flow analysis
[4] Alon, Galil, Margalit [5]; и Yuval [56]
[5] Bidirected graph

если в нём для каждого ребра $\langle u, v, (_i \rangle$ найдётся противоположное ребро $\langle v, u,)_i \rangle$ и наоборот.

Неформально, матрица смежности такого графа симметрична, и метки на симметричных рёбрах — это парные открывающая/закрывающая скобки.

TODO: картиночька

В своей работе Юань и др. [55] заметили, что при решении задачи анализа указателей[1] граф получается двунаправленным (так как отношения `GetField` и `PutField` взаимообратные). Также, они конструируют алгоритм, с временем работы $\mathcal{O}(|V| \log |V| \log k)$, в случае, если полученный граф является деревом. На внешнем уровне алгоритма строится центроидная декомпозиция, а для каждого конкретного центроида — бор S-префиксов[2] его поддерева. Чжан и др. [19], заметив, что в двунаправленных графах Диково достижимые вершины формируют классы эквивалентности (3.1), улучшили этот алгоритм, получив линейное время работы, а также сконструировали алгоритм для произвольных графов с временем работы $\mathcal{O}(|E| \log |E|)$. Позднее, Чаттерджи и др. [10] улучшили этот результат до $\mathcal{O}(|E| + |V|\alpha(|V|))$[3].

Если к задаче анализа указателей добавить контекстную чувствительность[4], то есть кроме записи/чтения полей учитывать ещё и вызов/возврат из функций, то её можно сформулировать в терминах смешанной Диковой достижимости.

Определение 1.4. Для двух языков L_1 и L_2, заданных над алфавитами Σ_1 и Σ_2 соответственно, определим *оператор смешения*[5] [29] $\odot : L_1 \times L_2 \to (\Sigma_1 \cup \Sigma_2)^*$ следующим образом:

- $a \odot \varepsilon = \{a\}$, где $a \in L_1$
- $\varepsilon \odot b = \{b\}$, где $b \in L_2$
- $c_1 a \odot c_2 b = \{c_1 w \mid w \in (a \odot c_2 b)\} \cup \{c_2 w \mid w \in (c_1 a \odot b)\}$,
  где $a \in L_1, b \in L_2, c_1 \in \Sigma_1, c_2 \in \Sigma_2$

Можно также переопределить оператор смешения для двух языков:
$L_1 \odot L_2 = \bigcup\limits_{a \in L_1, b \in L_2} a \odot b.$

Определение 1.5. Пусть есть два языка Дика $\mathcal{D}_i$ и $\mathcal{D}_j$, заданные над разными алфавитами. Тогда назовём язык $\mathcal{D}_i \odot \mathcal{D}_j$ *смешанным языком Дика*[6].

Неформально, это множество таких скобочных последовательностей, что их проекции на алфавиты Σ_i и Σ_j принадлежат $\mathcal{D}_i$ и $\mathcal{D}_j$ соответственно.

[1] Points-to Analysis

[2] Строк, сформированных наивным алгоритмом проверки правильности скобочной последовательности с использованием стека

[3] $\alpha(n)$ — обратная функция Аккермана

[4] Context sensitivity

[5] Interleaving operator

[6] Interleaved Dyck language или InterDyck language

Например, пусть $\mathcal{D}_b$ — языка квадратных ПСП и $\mathcal{D}_p$ — языка круглых ПСП. Тогда смешанный язык $\mathcal{D}_b \odot \mathcal{D}_p$ содержит такие слова как “([])” и “([()()])[]”.

Однако, задача смешанной Диковой достижимости в общем случаем неразрешима [40]. Более того, она неразрешима уже для языка $\mathcal{D}_2 \odot \mathcal{D}_2$ даже на двунаправленных графах. Поэтому Ли и др. [29] рассмотрели задачу $\mathcal{D}_1 \odot \mathcal{D}_1$-достижимости (на двунаправленных графах) и построили для неё алгоритм с временем работы $\mathcal{O}(|V|^7)$. Решение этой задачи можно использовать как приближение для более общей $\mathcal{D}_k \odot \mathcal{D}_k$-достижимости.

1.4. Выводы и результаты по главе

TODO

2. Алгоритм, основанный на пересечении языков

В качестве единого подхода к построению частичных решений для задачи CFPQ был выбран алгоритм [13, 16], основанный на пересечении языков. Причина выбора именно этого решения следующая: в основе остальных подходов лежат алгоритмы парсинга, все из которых имеют кубическое время работы и неизвестно, могут ли быть ускорены, тогда как данный подход основан на графовых алгоритмах (а именно, на решении задачи построения инкрементального транзитивного замыкания), засчёт чего более подвержен модификациям.

2.1. Сведение к задаче достижимости для РКА

Главной идеей алгоритма является следующее замечание: любой помеченный граф G можно рассматривать как *недетерминированный конечный автомат*, в котором не обозначены начальное и конечные состояния.

Определение 2.1. *Недетерминированный конечный автомат (или НКА)*[1] — это пятёрка $\langle Q, \Sigma, \delta, q_0, F \rangle$:

- Q — конечное множество состояний
- Σ — конечный алфавит
- $\delta \colon Q \times \Sigma \to 2^Q$ — функция перехода
- $q_0 \in Q$ — начальное (стартовое) состояние
- $T \subseteq Q$ — множество конечных (терминальных) состояний

Язык, распознаваемый НКА $\mathcal{A}$ — язык $L(\mathcal{A})$ слов, на которых автомат, следуя функции перехода, может дойти из стартового состояние в терминальное хотя бы одним способом.

Недетерминированные конечные автоматы задают *регулярные языки*.

TODO: пример (в фигуру с автоматом $0 \rightsquigarrow 2$)

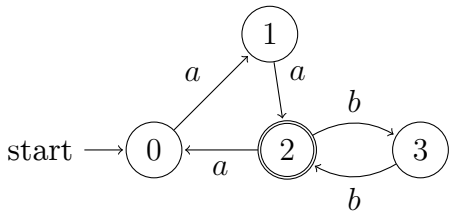


Рис. 3: НКА, задающий слова, читаемые на путях $0 \rightsquigarrow 2$.
Например, *aa*, *aabb* или *aabbaaa*

[1] Nondeterministic finite automaton (или NFA)

Утверждение 2.1. *Если зафиксировать конкретные вершины u и v помеченного графа G как стартовое и конечное состояния, то полученный автомат будет задавать язык слов, читаемых на путях из u в v.*

Язык, задаваемый подобным автоматом, будем обозначать как $L(G, u, v)$.

Получаем, что для каждой пары вершин u и v, наличие пути $p\colon u \rightsquigarrow v$, такого, что на нём читается слово $w \in L(\mathcal{G})$ равносильно непустоте пересечения языков $L(G, u, v)$ и $L(\mathcal{G})$ (действительно, в пересечении будут лежать ровно слова, выводимые грамматикой и при этом читаемые на каком-либо пути $u \rightsquigarrow v$).

Для построения пересечения языков нам потребуется такая конструкция, как прямое произведение автоматов.

Определение 2.2 (Прямое произведение автоматов)**.** **TODO**

Утверждение 2.2. *[24] Автомат A, построенный как прямое произведение автоматов A_1 и A_2 ($A = A_1 \otimes A_2$), распознаёт язык, равный пересечению языков A_1 и A_2 ($\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$)*

Заметим, однако, что лишь один из языков, чьё пересечение нам нужно, является регулярным (и, следовательно, задаётся автоматом) — язык входного графа. Второй же язык является контекстно-свободным и недетерминированный конечным автоматом не задаётся. Однако он задаётся с помощью более сложного автомата — рекурсивного.

Определение 2.3 (Рекурсивный конечный автомат (РКА))**.** *Рекурсивный конечный автомат (или РКА)*[1] [6] — это набор компонент $\langle M_1, M_2, \ldots, M_k \rangle$, с выделенной стартовой компонентой, где каждая компонента M_i — это пятёрка $\langle Q_i, \Sigma_i, En_i, Ex_i, \delta_i \rangle$:

- Q_i — конечное множество состояний
- Σ_i — конечный алфавит
- $En_i \subset Q_i$ — множество начальных состояний
- $Ex_i \subset Q_i$ — множество конечных состояний
- $\delta_i\colon Q_i \times (\Sigma_i \cup \bigcup\limits_{j=1}^{k} En_i \times Ex_i) \to 2^{Q_i}$ — функция перехода. У δ_i есть два типа переходов: *внутренние*, которые работают как обычные переходы в НКА и *рекурсивные*, которые делают вызов другой компоненты (при этом обозначая начальную и конечную вершину в ней).

Неформально, это набор компонент, каждая из которых представляет собой НКА, на рёбрах которого могут быть “рекурсивные вызовы” других компонент.

TODO: пример

[1] Recursive state machine

Утверждение 2.3. *[6] Утверждение 2.2 остаётся верным, если один из языков задан РКА.*

При прямом произведении НКА и РКА получается также РКА. А именно, при произведении РКА $\mathcal{R} = \langle M_1, \ldots, M_k \rangle, M_i = \langle Q_i, \Sigma_i, En_i, Ex_i, \delta_i \rangle$ и НКА $\mathcal{A} = \langle Q, \Sigma, \delta, q_0, F \rangle$, получается РКА $\mathcal{P} = \mathcal{R} \otimes \mathcal{A}$, состоящий из k компонент $\mathcal{M}_i$, таких что

- множество состояний $\mathcal{M}_i$ равно декартову произведению $Q_i \times Q$
- множество начальных состояний $\mathcal{M}_i$ равно $En_i \times Q$
- множество конечных состояний $\mathcal{M}_i$ равно $Ex_i \times Q$
- для каждого внутреннего перехода $q_s \xrightarrow{a} q_t \in \delta_i$, в $\mathcal{M}_i$ есть внутренние переходы $\langle q_s, u \rangle \xrightarrow{a} \langle q_t, v \rangle$ для всех $u, v \in Q$, что $u \xrightarrow{a} v \in \delta$
- для каждого внешнего перехода $\langle q_s \xrightarrow{en,ex} q_t \rangle \in \delta_i$, в $\mathcal{M}_i$ есть внешние переходы $\langle q_s, q \rangle \xrightarrow{\langle en,q \rangle, \langle ex,q \rangle} \langle q_t, q \rangle$ для всех $q \in Q$

TODO: сделать на него ссылки везде (не только на него)

Все алгоритмы для CFPQ, которые будут описаны в этой работе имеют следующую схему:

1. Построим прямое произведение входной грамматики $\mathcal{R}$ и входного графа G: $\mathcal{P} = \mathcal{R} \otimes G$.
2. *Решим задачу достижимости для полученного РКА $\mathcal{P}$*
3. Из вершины u в вершину v входного графа существует путь, выводимой входной грамматикой $\mathcal{G} \Leftrightarrow$ в $\mathcal{P}$ есть путь из стартового состояния (q_0, u) в конечное состояние (q_f, v)

Рассмотрим внимательнее второй пункт — задачу достижимости для РКА. В случае обычного автомата эта задача эквивалентна задаче построения транзитивного замыкания [51]. В случае же РКА задача осложняется наличием рекурсивных вызовов, которые разрешаются итеративно. (??)

2.2. Алгоритм П

В листинге 1 приведён псевдокод Алгоритма П.

TODO: что-то написать про епс-переходы

Listing 1 Алгоритм достижимости для РКА

```
1:  function RSMReachability(R)
2:      A ← Adjacency matrix for R
3:      while A is changing do
4:          A' ← transitiveClosure(A)              ▷ Построение транзитивного замыкания
5:          for i ∈ 1..k do
6:              for u ∈ En_i do
7:                  for v ∈ Ex_i do
8:                      if A'_{u,v} ∧ ¬A_{u,v} then
9:                          A' ← A' ∪ getEdges(i, u, v)        ▷ Добавление новых рёбер
10:         A → A'
11:     return A
```

Работа происходит над матрицей смежности $\mathcal{R}$ — изначально туда записываются все “внутренние” (нерекурсивные) рёбра.

Далее, внешний цикл повторяется, пока матрица смежности A меняется (т.е. пока добавляются новые рёбра). На каждой итерации считается A' — транзитивное замыкание A. После этого находятся все новые пути вида $\langle$стартовое состояние$\rangle \rightsquigarrow \langle$конечное состояние$\rangle$ — те рёбра между стартовой и конечной вершинами компоненты, которых не было в A, но которые есть в A' — и добавляются соответствующие этим путям рёбра: для нового пути $(u \in En_i) \rightsquigarrow (v \in Ex_i)$ проводятся все рёбра, соответствующие рекурсивным вызовам i-ой компоненты с начальной вершиной u и конечной вершиной v.

2.2.1. Пример

TODO

2.2.2. Время работы

Время работы — $k \cdot T(n)$, где k — число итераций внешнего цикла, $T(n)$ — время работы одной итерации.

Оценим $T(n)$. Внутренняя часть цикла состоит из двух частей: нахождения транзитивного замыкания (строка 4) и прохода по матрице для выявления новых рёбер (строки 5-9).

Задача поиска транзитивного замыкания эквивалентна задаче перемножения булевых матриц [2] и может быть решена сведением к быстрому перемножению (обычных) матриц за $\mathcal{O}(n^\omega)$, где $2 < \omega < 2.273$ [4].

Проход по матрице (строки 5-7) работает за $\mathcal{O}(n^2)$, что доминируется временем построения транзитивного замыкания. Добавление новых рёбер (строки 8-9) отработает

суммарно за $\mathcal{O}(n^2)$ (т.к. каждое ребро будет добавлено не более одного раза).

Итого, время работы алгоритма $\mathcal{O}(k \cdot n^\omega)$.

2.3. Алгоритм П2

Можно заметить, что не очень осмысленно на каждой итерации заново считать транзитивное замыкание, достаточно искать только пути, проходящие через рёбра, добавленные непосредственно на предыдущей итерации. То есть достаточно решать задачу **инкрементального** транзитивного замыкания.

В листинге 2 приведён псевдокод Алгоритма П2 (основанного на инкрементальном ТЗ)

Listing 2 Алгоритм достижимости для РКА (2)

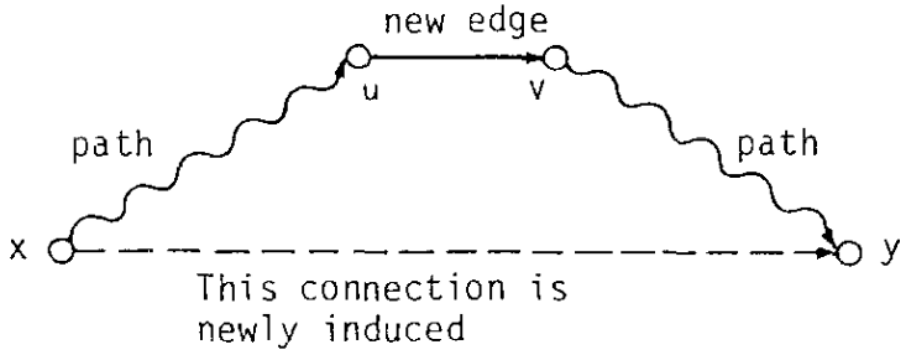
```
1:  function RSMReachability2(𝓡)
2:      A ← Empty adjacency matrix
3:      Q ← Empty Queue
4:      for i ∈ 1..k do
5:          for u -c-> v ∈ δ_i do
6:              Q.Push(⟨u, v, i⟩)
7:      while Q is not Empty do
8:          ⟨u, v, i⟩ ← Q.Pop()
9:          if u ∈ En_i ∧ v ∈ En_i then                ▷ Нашли новый путь
10:             A ← A ∪ getEdges(i, u, v)
11:             Q.PushAll(getEdges(i, u, v))           ▷ Добавляем новые рёбра
12:         for x ∈ Q_i do
13:             if A_{x,u} ∧ ¬A_{x,v} then
14:                 for y ∈ Q_i do
15:                     if A_{v,y} ∧ ¬A_{x,y} then
16:                         A ← A ∪ ⟨x, y⟩
17:                         Q.Push(⟨x, y, i⟩)          ▷ Обновлем транзитивное замыкание
18:     return A
```

В алгоритме используется (более менее) стандартная реализация инкрементального транзитивного замыкания [25]. Для этого в ходе работы алгоритма поддерживается рабочая очередь Q рёбер транзитивного замыкания, которые были найдены, но ещё не обработаны.

При обработке очередного *(потому что оно из очереди ахахах)* ребра, ищутся новые пути, которые проходя через него. А именно, пусть было добавлено ребро $u \to v$. Тогда далее перебирается вершина x, такая что из неё была достижима вершина u

$(x \rightsquigarrow u)$, но не была достижима вершина v $(x \not\rightsquigarrow v)$. Из такой вершины x становятся достижимы все вершины y, которые были достижимы из v $(v \rightsquigarrow y)$.

Также, как и в Алгоритме П, если ребро ТЗ (= путь в графе) соединяет начальную и конечную вершину, в очередь добавляются также все соответствующие ему рекурсивные рёбра.



TODO: норм картинка

2.3.1. Время работы

Внешний цикл итерируется по всем рёбрам, так что работает за $\mathcal{O}(m^*)$. Добавление новых рёбер как и в алгоритме П рассмотрит каждое ребро не более одного раза, так что тоже работает за $O(m^*)$. Нужно только оценить работу по поддержанию транзитивного замыкания.

Цикл на строке 12 перебирает все вершины, так что отработает суммарно за $\mathcal{O}(m^*n)$. Внутренний цикл (строка 14) тоже перебирает все вершины, но после его выполнения будет добавлено хотя бы одно новое ребро $(x \to v)$, так что суммарное время работы этих циклов также можно оценить как $\mathcal{O}(m^*n)$.

Итого, суммарное время работы составляет $\mathcal{O}(nm^*)$, что в плотных графах будет равно $\Theta(n^3)$.

Замечание. Время работы итеративного транзитивного замыкания можно ускорить в $\log n$ раз [13], воспользовавшись методом четырёх русских [35] (так как работа происходит над булевыми векторами).

2.4. Загадка дыры

Основой для получения частных решений будут модификации Алгоритмов П и П2.

Модифицируем Алгоритм П2

Как уже было сказано, узким местом Алгоритма П2 является построение инкрементального транзитивного замыкания, которое в общем случае нельзя (скорее всего) решить быстрее, чем за кубическое время.

Следовательно, чтобы получить более быстрый алгоритм в частном случае, нужно рассматривать такие частные случаи, для которых задачу инкрементального транзитивного замыкания можно решать быстрее.

Рассмотрим сначала всякие несложные случаи:

- Графы с ограниченной степенью

  В [53] представлен алгоритм для инкрементального транзитивного замыкания на графах с ограниченной исходящей степенью. Асимптотика алгоритма: $\mathcal{O}(dm^*)$, где d — ограничение сверху на исходящую степень графа, а m^* — число рёбер в транзитивном замыкании.

- Планарные (?)

  TODO: Разобраться, что там написала Александра

- Bounded-stack SM

- Неориентированные графы

  Вот тут получаем нетривиальные результаты, про них в следующем разделе

2.5. Выводы и результаты по главе

TODO

3. Результаты (1)

Для неориентированных графов отношение достижимости симметрично и на самом деле это отношение “принадлежать одной компоненте связности”. Поддерживать добавление рёбер и проверку связности в неориентированном графе может СНМ.

3.1. Алгоритм, основанный на неориентированном транзитивном замыкании

Я буду называть его Алгоритм НП

В листинге 3 приведён псевдокод Алгоритма НП.

Listing 3 Алгоритм достижимости для РКА, основанный на неориентированном ТЗ

```
1: function UndirectedRSMReachability(R)
2:     A ← Empty adjacency matrix
3:     Q ← Empty Queue
4:     D ← DSU(| ⋃_{i=1}^{k} Q_i |)
5:     for i ∈ 1..k do
6:         for u -c-> v ∈ δ_i do
7:             Q.Push(⟨u, v, i⟩)
8:     while Q is not Empty do
9:         ⟨u, v, i⟩ ← Q.Pop()
10:        if u ∈ En_i ∧ v ∈ En_i then                ▷ Нашли новый путь
11:            A ← A ∪ getEdges(i, u, v)
12:            Q.PushAll(getEdges(i, u, v))
13:            D.Union(u, v)                          ▷ Добавляем новые рёбра
14:    return A
```

Для реализации алгоритма используются две вспомогательные структуры данных: очередь Q, хранящая рёбра, которые были добавлены в граф, но ещё не обработаны (как и в оригинальном алгоритме П2), и СНМ D, поддерживающая компоненты связности и поиск новых путей $\langle$стартовое состояние$\to$конечное состояние$\rangle$.

Опишем подробно структуру используемого СНМ (в листинге 4 приведён псевдокод.

Listing 4 Система Непересекающихся Множеств

```
1:  Structure DisjointSets
2:      function DisjointSets(V)
3:          for v ∈ V do
4:              P[v] ← v                                          ▷ Предкок
5:              R[v] ← 0                                          ▷ Ранг
6:          for v ∈ En(V) do
7:              En[v] ← {v}                          ▷ Список стартовых вершин поддерева
8:          for v ∈ Ex(V) do
9:              Ex[v] ← {v}                          ▷ Список конечных вершин поддерева
10:     function Find(v)
11:         if P[v] = v then return v
            return P[v] = Find(P[v])                  ▷ Эвристика сжатие путей
12:     function Union(u, v)
13:         u ← Find(u)
14:         v ← Find(v)
15:         if u = v then return
16:         if R[u] > R[v] then
17:             Swap(u, v)                                    ▷ Ранговая эвристика
18:         Q.PushAll({⟨en_u, ex_v⟩ | en_u ∈ En[u], ex_v ∈ Ex[v]})
19:         Q.PushAll({⟨en_v, ex_u⟩ | en_v ∈ En[v], ex_u ∈ Ex[u]})   ▷ Добавление новых
    рёбер
20:         En[v] ← En[v] ∪ En[u]
21:         Ex[v] ← Ex[v] ∪ Ex[u]
22:         R[v] ← max(R[v], R[u] + 1)
23:         P[u] = v                                          ▷ Объединение компонент
24:
```

За основу взята стандартная реализация [23] на подвешенные деревья, использующая обе эвристики: сжатие путей и ранговую.

Дополнительно в корнях хранятся списки всех начальных и конечных состояний компоненты. При добавлении ребра в операции *Join* перебираются все пары начальная/конечная вершина из двух компонент и соответствующие им рёбра добавляются в рабочую очередь Q.

TODO: (подумать) можно ли добавлять сразу много рёбер и сжимать их дфсом (как Борувка)?

3.2. Время работы

Теорема 3.1. *На РКА из n состояний и m^* рёбрах в транзитивном замыкании, Алгоритм НП отработает за время $\mathcal{O}(n + m^*\alpha(m^* + n, n))$*

Доказательство.
TODO: *m^* джойнов, а проходы по спискам не долгие, так как каждый раз генерируем новое ребро* □

3.3. Корректность для неориентированных графов и некоторых классов грамматик

TODO: может, в мусорку этот subsection?

3.4. Корректность для двунаправленных графов и языка Дика

Напоминание, что для языка Дика контекстно-свободная достижимость → Дикова достижимость

Для доказательства потребуется следующее вспомогательное утверждение

Лемма 3.1. *Для вершин двунаправленных графов отношении Диковой достижимости является отношением эквивалентности.*

Доказательство.
TODO: ну тут тупо

□

Замечание. Для данного алгоритма будем использовать следующий вид грамматики для языка Дика:

$S \to \varepsilon \mid SS \mid ({}_1S)_1 \mid \ldots \mid ({}_kS)_k$

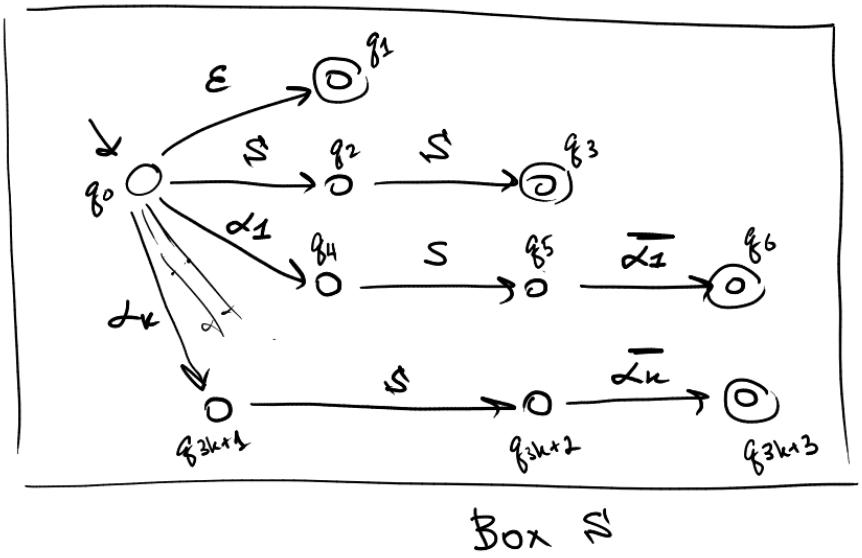


Рис. 4: РКА для языка Дика

На рисунке 4 приведена РКА для данной грамматики. Заметим, что он содержит всего одну компоненту (S).

TODO: нарисовать красиво

Теорема 3.2. *Решение для CFPQ, использующее Алгоритм НП, работает корректно на двунаправленных графах и языке Дика.*

Доказательство. Достаточно доказать, что для любой пары состояний $u \in En_i, v \in Ex_i$ существование неориентированного пути эквивалентно существованию ориентированного.

$\Leftarrow$ (ориентированный $\Rightarrow$ неориентированный)

Очевидно, если есть ориентированный путь $u \rightsquigarrow v$, то ровно если убрать ориентацию этот путь никуда не денется.

$\Rightarrow$ (неориентированный $\Rightarrow$ ориентированный)

At first, note that the Kronecker product $G \otimes \mathcal{G}$ forms some kind of a layered structure — i-th layer consists of vertices (q_i, v), where q_i is i-th RSM state. Because RSM is topologically sorted (TODO), every edge $(q_i, u) \to (q_j, v)$ goes forward.

We will call path *simple* if it visits every layer no more than once.

We prove the claim by induction on the l (path length).

Clearly the result is true for $l \leqslant 3$, because the only way to achive final vertex in $1, 2$ or 3 edges is by a simple vertical path (which exists in the original graph too).

Otherwise (if $l \geqslant 4$), path is not simple.

Consider the first flex point of the path, that is the vertex (q_i, v) such that edges $(q_j, u) \to (q_i, v)$ and $(q_i, v) \to (q_k, w)$ are in the path and $j, k \leqslant i$ (so, the path is convex at this point).

Looking at the grammar graph we can notice, that every state has indegree $\leqslant 1$. So at the flex point there are actually two same-labeled edges (that is, $j = k$).

There can be three different types of labels on those edges:

- α_l-label

  path: $(q_0, u) \to (q_i, v) \to (q_0, w) \to \cdots \to (q_f, z)$.

  Since α_l-labeled edges could only be added on the initialization stage, graph $\mathcal{G}$ contains edges $u \xrightarrow{\alpha_l} v$ and $w \xrightarrow{\alpha_l} v$. Notice, that cause $\mathcal{G}$ is bidirected, it also has to contain edges $v \xrightarrow{\overline{\alpha_l}} u$ and $v \xrightarrow{\overline{\alpha_l}} w$.

  Now we can notice, that w is Dyck-reachable (by the path $\alpha_l\overline{\alpha_l}$) from u, so there is an S-labeled edge from u to w. We can also conclude, that (by induction) there is a directed path from (q_0, w) to (q_f, z) (there z is the end of the path and q_f is some final state of G), so there is an S-labeled edge from w to z.

  Using this two observation we can construct a directed path from u to z: $u \xrightarrow{S} w \xrightarrow{S} z$.

- S-label

  path: $(q_0, a) \to \cdots \to (q_j, u) \to (q_i, v) \to (q_j, w) \to \cdots \to (q_f, z)$.

  $\mathcal{G}$ contains S-labeled edges $u \xrightarrow{S} v$ and $w \xrightarrow{S} v$. Since $\mathcal{G}$ is bidirected, then by **??** $v \xrightarrow{S} u$ and $v \xrightarrow{S} w$. Combining $u \xrightarrow{S} v$ and $v \xrightarrow{S} w$ we get that $u \xrightarrow{S} w$.

  No we want to sort of contract this edge, joining u and w (on the j-th level). Then we can get (by induction hypothesis) the directed path from $(q_0, a) \rightsquigarrow (q_f, z)$. If new path does not contain joined uw vertex, then that's the answer. Otherwise we can split this vertex back, inserting between u and w the S-labeled path (that one, from $u \xrightarrow{S} w$ edge). We can do it, because the both of these paths form correctly matched parenthesis (TODO: we can prove this using stack-based checking algorithm).

  Two other cases can be proved the same way, but I find it a little dishonest

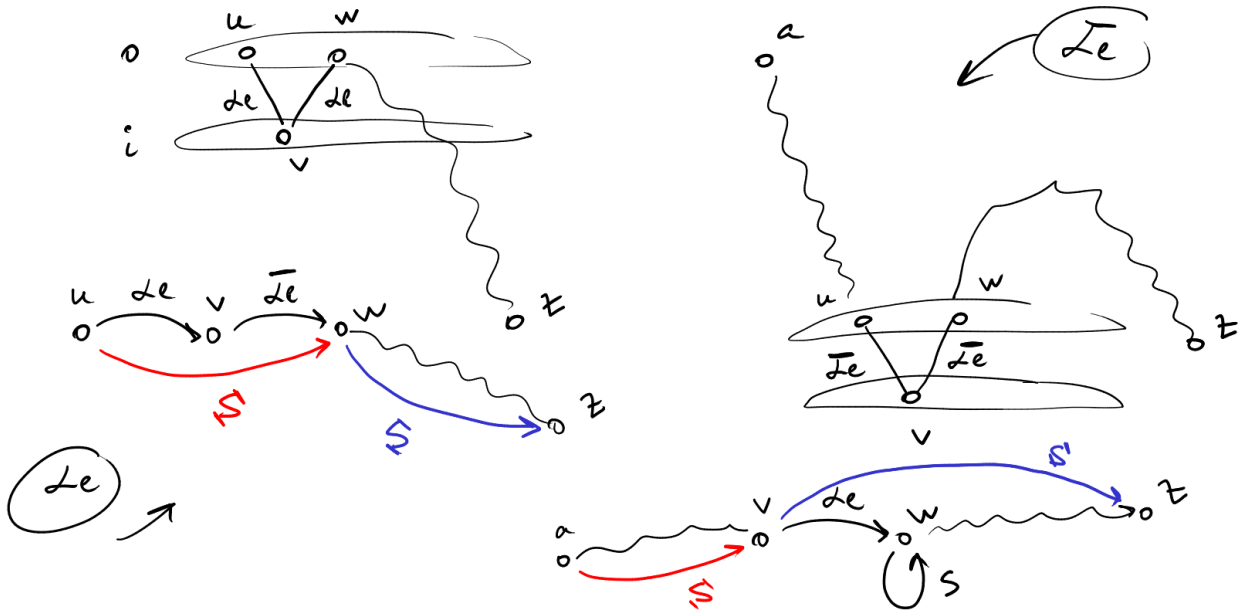
- $\overline{\alpha_l}$-label

  path: $(q_0, a) \to \cdots \to (q_j, u) \to (q_f, v) \to (q_j, w) \to \cdots \to (q_f, z)$.

  Since α_l-labeled edges could only be added on the initialization stage, graph $\mathcal{G}$ contains edges $u \xrightarrow{\overline{\alpha_l}} v$ and $w \xrightarrow{\overline{\alpha_l}} v$. Notice, that cause $\mathcal{G}$ is bidirected, it also has to contain edges $v \xrightarrow{\alpha_l} u$ and $v \xrightarrow{\alpha_l} w$.

  By induction, we get that $a \xrightarrow{S} v$. Now we will construct a second part of the path: $(q_0, v) \xrightarrow{\alpha_l} (q_{j-1}, w) \xrightarrow{S} (q_j, w) \rightsquigarrow (q_f, z)$ $(q_{j-1}, w) \xrightarrow{S} (q_j, w)$ — initial S-loop). By induction, we have directed simple version of this path, so $v \xrightarrow{S} z$.

  Combining this two paths ($a \xrightarrow{S} v$ and $v \xrightarrow{S} z$) we get $a \xrightarrow{SS} z \Rightarrow a \xrightarrow{S} z$ — desired path.



□

3.5. Выводы и результаты по главе

4. Результаты (2)

4.1. Алгоритм для языка Дика на одном типе скобок

Замечание. Строки, принадлежащие языку Дика $\mathcal{D}_1$ часто изображают в виде так называемых *путей Дика* — путей из точки $(0, 0)$ в точку $(0, 2n)$, не опускающихся ниже оси абсцисс. Открывающей скобке соответствует вектор $(1, 1)$, закрывающей — $(1, -1)$.

TODO: Картинка

TODO: для semi-dyck тоже работает, нужно это упомянуть

Для построения алгоритма воспользуемся следующим результатом:

Лемма 4.1 (Дюлеаж и Лоран [17])**.** *Для языка L определим $p_L(n)$[1] — максимальная длина кратчайшего слова в $L \cap K$ по всем регулярным языкам K, задаваемым НКА с $\leqslant n$ состояниями.*

Тогда для языка Дика D_1 на одном типе скобок $p_{D_1}(n) = \mathcal{O}(n^2)$[2]

Следствие 4.0.1. *Для любой пары вершин $u, v \in V(G)$, если есть Диков путь $u \rightsquigarrow v$, то существует и Диков путь $u \rightsquigarrow v$, длина которого $\mathcal{O}(n^2)$.*

Доказательство. Слова, читаемые на путях $u \rightsquigarrow v$ задаются НКА на n вершинах — графом G, в котором u и v выбраны за начальное и конечное состояния соответственно. □

Замечание. Пользуясь этим фактом, можно соорудить наивный алгоритм — достаточно лишь заметить, что раз длина искомого пути всегда ограничена, то можно задать такие пути с помощью ДКА: язык D_1 задаётся автоматом с одним счётчиком, значение счётчика не превышает $\mathcal{O}(n^2)$, так что его можно закодировать в состояние. Однако и размер такого ДКА будет $\mathcal{O}(n^2)$, что для данной задач слишком много.

TODO: дописать красиво

Ну тут мотивация простая, если строить автомат по грамматике, то часто он получается экспоненциального размера. Сейчас хотим то же в обратную сторону провернуть. Ещё мотивация: чтобы хранить одну циселку порядка n^2 в считающем автомате нужно всего $\mathcal{O}(\log n)$ бит, так что хочется в такое ограничение и уложиться

TODO: написать красивые слова

Хочется тут красивую подводку, но не выходит...

Хотим использовать Алгоритм П (потому что иначе субкубическое решение не получим), так что нужно, чтобы у нас было мало итераций. С обычной грамматикой

[1] Формально, $p_L(n) = \max\{\min\{|w| \colon w \in L \cap K\}, K \in Rat_n(X), L \cap K \neq \varnothing\}$,
где $Rat_n(X)$ — регулярные языки над алфавитом X, распознаваемые НКА с $\leqslant n$ состояниями.

[2] Точная оценка — $2n^2 + 4n$

$(S \to \varepsilon \mid (S) \mid SS)$ ничего хорошего не выйдет: рассмотрим путь $(^k)^k$, чтобы его найти потребуется k итераций. Так что наша цель — за малое число итераций (добавления рёбер) находить такие длинные пути.

Строим РКА, компоненты описываются в виде продукций, обсуждалось (**TODO**: пока что ещё нет), как строить одно из другого.

Строим $2K$ (где $K = \lceil \log(2n^2 + 4n) \rceil$) компонент РКА, отвечающих за сжатие вертикальных путей: U_i сжимают пути вида $(^{2^i}$, D_i сжимают пути вида $)^{2^i}$:

$U_0 \to ($

$U_1 \to U_0 S U_0$

$U_2 \to U_1 S U_1$

...

$U_K \to U_{K-1} S U_{K-1}$

В продукции для U_i есть S между U_{i-1} и U_{i-1} — она нужна, так как мы ищем не строго возрастающие пути, а допускаем им некоторое время "идти прямо" (получаются уже такие пути Моцкина, а не Дика, в которых S соответствуют горизонтальным рёбрам)

Продукции для D_i выглядят аналогично.

Теперь можем строить компоненту, отвечающую за, собственно, пути Дика. Продукция $S \to (S)$ в обычной грамматике как бы "сжимала уголки". Теперь, пользуясь U_i и D_i можем "сжимать уголки" побольше.

$S \to \varepsilon \mid U_0 S D_0 \mid U_1 S D_1 \mid \ldots \mid U_K S D_K \mid S^*$

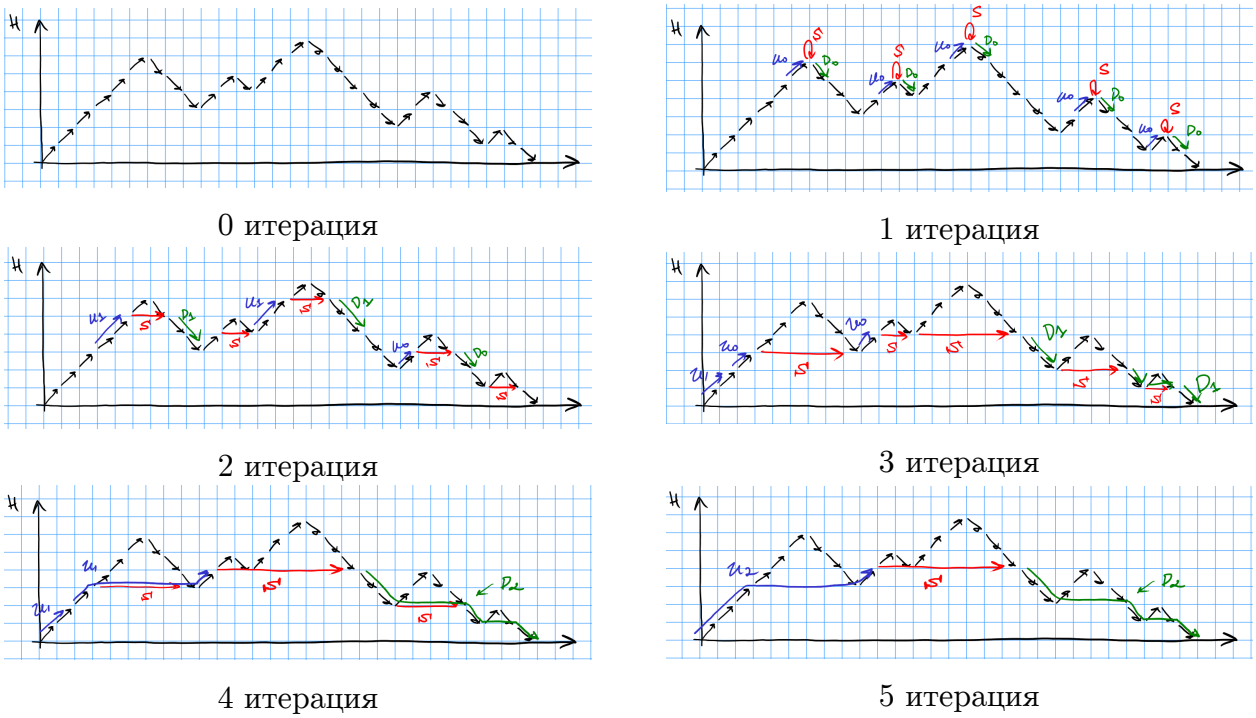
Продукция $S \to S^*$ обозначает петлю на терминальной вершине в компоненте S в РКА грамматики (это так называемая, продукция транзитивного замыкания, её предназначение — да одну итерацию сжимать все пути из S-ок; в произведении с графов она образует клику на терминальных состояних в компоненте).

При оценке асимптотики алгоритма будет важно, что все компоненты РКА имеют константный размер (число состояний), так что разобьём компоненту S на K штук: $S_i \to U_i S D_i$, $S \to (S_0 \mid S_1 \mid \ldots \mid S_k)^*$ (одно стартовое/терминальное состояние, на котором весит K петель).

Собственно, алгоритм — запустить алгоритм П (**TODO**: линк) на прямом произведении входного графа и описанной выше грамматики.

TODO: *можно сюда просто вставить маленький пример, где рассматривается один путь и показано, как он сжимается, без пересечения с грамматикой.*

4.2. Пример



0 итерация

1 итерация

2 итерация

3 итерация

4 итерация

5 итерация

Рис. 5: Пример работы алгоритма (на конкретном пути)

4.3. Корректность алгоритма

TODO

4.4. Время работы

TODO: *Я сейчас докажу* $\mathcal{O}(n^\omega \log^3 n)$*, но, кажется, можно сократить на лог (засчёт того, что у нас параллельно сжимаются* S*-ки, и* U*-шки с* D*-шками)*

Для доказательства времени работы нам потребуется пара вспомогательных утверждений.

Лемма 4.2. *После* i*-ой итерации алгоритма*

Теорема 4.1. *Время работы Алгоритма (link (?)) на языке Дика на одном типе скобок —* $\mathcal{O}(n^\omega \log^3 n)$*, где* n *— число вершин входного графа.*

Доказательство. Время работы алгоритма П состоит из двух множителей: времени работы одной итерации и числа итераций.

Время работы одной итерации в алгоритме П составляло $\mathcal{O}(N^\omega)$ (где N — число состояний РКА-произведения), т.к. на каждой итерации считалось транзитивное замыкание. Заметим, что на самом деле достаточно находить транзитивное замыкание отдельно в каждой компоненте РКА (т.к. разные компоненты вообще не

связны). Т.к. компоненты РКА грамматики константны, то размер компонент РКА-произведения — $\mathcal{O}(n)$. Так что суммарное время работы одной итерации составит $\mathcal{O}(n^\omega K)$.

Оценим теперь число итераций.

TODO: *за лог итераций срезаются все уголки (локальные максимумы), нужно теперь их аккуратно посчитать, смотря на соседей (и минимумы)*

□

Замечание. Этот результат не обобщается на языки Дика с большим типом скобок.

Мотивация примерно такая: во1, они сложнее (язык Дика на $\geqslant 2$ типах скобок — такой же мощный как и произвольный КС-язык (теорема Хомского-Шутценбергера, тогда как язык Дика на одном типе скобок вроде как попроще. *Ещё Дик на $\geqslant 2$ типах скобок генерирует full AFL (Abstract Families of Languages) (что бы это не значило)*), во2, сейчас мы играли на том, что состояние — примерно одна чиселка (ну опять-таки, язык Дика на одном типе скобок распознаётся автоматом с одним счётчиком), а для большего типа скобок нужен весь стек.

4.5. Выводы и результаты по главе

5. Результаты (3)

5.1. Алгоритм для смешанного языка Дика

Определение 5.1. **TODO**: нормально определение в секции с определениями

Пусть у нас есть два языка Дика D_i и D_j над разными алфавитами (например, в одном — $(_i,)_i$, а в другом $[_i,]_i$). Тогда в смешанном языке Дика $D_i \bigodot D_j$ лежат слова, такие что, если рассмотреть проекции на алфавиты D_i и D_j, то это корректные слова из D_i и D_j (*Ужасно..*)

Пример: "([(])[[(])])" — корректное слово из $D_1 \bigodot D_1$.

Замечание. $D_i \bigodot D_j$ — это на самом деле пересечение двух КС языков (как обычные языки Дика, но там где ε, ещё можно скобки второго типа **TODO**).

Однако КС-языки не замкнуты относительно пересечения, более того, проверять непустоту пересечения двух КС-языков undecidable (**TODO**: link)

Лемма 5.1. *[29] В двунаправленном графе, если между парой вершин (u, v) существует какой-то $D_1 \bigodot D_1$ путь, то существует и такой путь, на котором в любой момент времени вложенность хотя бы одного типа скобок не превышает $6n$.*

В доказательстве крутят циклы, в подробности я не вдавалась

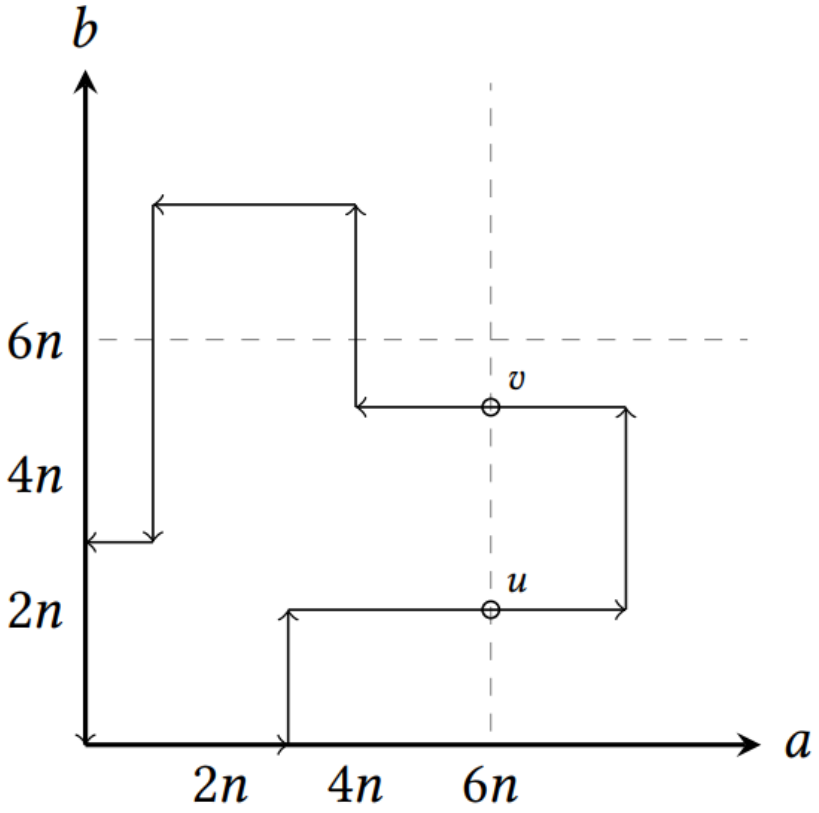


Рис. 6: Путь (кродеться)

Замечание. То есть путь можно искать следующего вида: он в основном ходит внутри квадрата $[0, 6n] \times [0, 6n]$, иногда вылезая из него, но только по одной координате.

Т.е. нужно разобрать отдельно две сущности: куски путей внутри квадрата, и куски путей, которые выходят погулять.

Вооружившись этим знанием, соорудим вот какой алгоритм:

1. **Часть про пути в квадратике**

   Помним (из прошлой главы), что если хотим решать D_1-достижимость, зная, что вложенность ограничена, можем это с помощью ДКА делать.

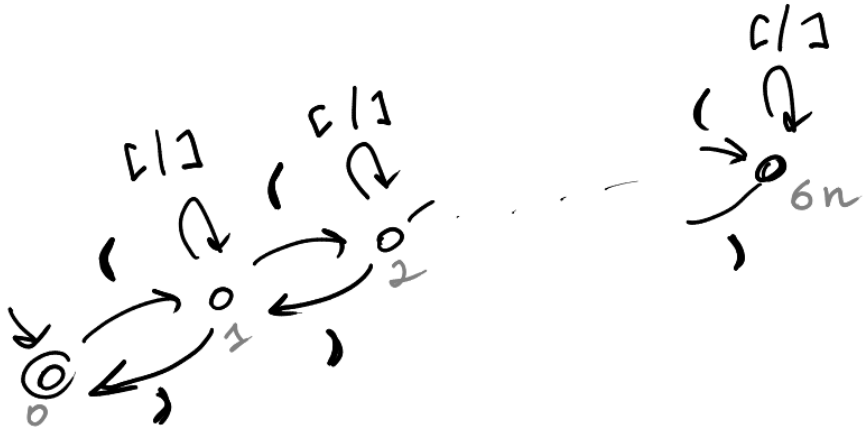


Рис. 7: ДКА для круглых скобок

   Строим такие ДКА для обоих типов скобок. Т.к. и тот, и другой регулярный, то можем их прекрасно пересечь с нашим графом.

   Получается такой граф троек (u, b, p) (вершина, баланс квадратных скобок, баланс круглых скобок) на $\mathcal{O}(n^3)$ вершинах и $\mathcal{O}(n^2 m)$ рёбрах.

   Ещё в нём будут ε-рёбра, найденные второй частью алгоритма. Эти рёбра имеют вид $(u, 6n, a) \to (v, 6n, b)$. Для всех возможных пар a/b и u/v их $\mathcal{O}(n^4)$.

   Т.к. входной граф двунаправленный, то полученное произведение будет неориентированным (*это не совсем тривиально, но понятно*), значит достижимость на нём можно искать просто dfs'ом за $\mathcal{O}(n^4)$.

2. **Часть про гуляющие пути**

   В какой момент пути идут гулять? Когда вложенность одной из скобок ровно $6n$. А когда возвращаются? Тоже когда ровно $6n$. А во все моменты между $\geqslant 6n$.

   То есть хотим искать такие пути, что по одной скобке они просто сбалансированы $(6n \rightsquigarrow 6n)$, а по другой какие попало, но всё время не больше $6n$ $(a \rightsquigarrow b, a, b \leqslant 6n)$.

Опять-таки, решаем через пересечение языков. Первый — наш граф (L_1). Второй — просто язык Дика (L_2) (опять-таки с оговорками про то, что скобочка другого типа — это то же, что и ε). Третий — пути Дика, начинающиеся в балансе a и заканчивающиеся в балансе b (L_3) (решаем для всех возможных пар балансов, чтобы рёбра провести). Давайте решать отдельно для каждого a, то есть нам нужен язык Диковых последовательностей со стартовым балансом a, таких, что их баланс не уходит в минус. Язык, опять-таки регулярный.

Дальше пересекаем. Сначала пересекаем $L_1 \cap L_2$. Это просто прямое произведение — граф пар (u, b) (для каждого a отдельно, снаружи всего это как бы фор по a). Вершин в нём $\mathcal{O}(n^2)$, рёбер $\mathcal{O}(nm)$. Теперь нужно это ещё пересечь с L_3. Ну так это же просто язык Дика, а Дикову достижимость мы за $\mathcal{O}(m^*\alpha)$ на двунаправленных графах решаем (а произведение $L_1 \cap L_2$ как раз двунаправленное). Получаем $\mathcal{O}(n \cdot nm^*\alpha) = \mathcal{O}(n^4\alpha)$.

5.2. Выводы и результаты по главе

Список литературы

[1] Abiteboul Serge, Hull Richard, Vianu Victor. Foundations of Databases. — Addison-Wesley, 1995. — ISBN: 0-201-53771-0.

[2] Aho Alfred V., Hopcroft John E. The Design and Analysis of Computer Algorithms. — 1st edition. — USA : Addison-Wesley Longman Publishing Co., Inc., 1974. — ISBN: 0201000296.

[3] Aho Alfred V, Hopcroft John E, Ullman Jeffrey D. Time and tape complexity of pushdown automaton languages // Information and Control. — 1968. — Vol. 13, no. 3. — P. 186–206.

[4] Alman Josh, Williams Virginia Vassilevska. A Refined Laser Method and Faster Matrix Multiplication. — 2020. — 2010.05846.

[5] Alon Noga, Galil Zvi, Margalit Oded. On the exponent of the all pairs shortest path problem // Journal of Computer and System Sciences. — 1997. — Vol. 54, no. 2. — P. 255–262.

[6] Analysis of Recursive State Machines / Rajeev Alur, Michael Benedikt, Kousha Etessami et al. — 2005. — Jul. — Vol. 27, no. 4. — P. 786–818. — Access mode: https://doi.org/10.1145/1075382.1075387.

[7] Barrett Chris, Jacob Riko, Marathe Madhav. Formal-Language-Constrained Path Problems // SIAM J. Comput. — 2000. — 01. — Vol. 30. — P. 809–837.

[8] Bradford Phillip G. Efficient exact paths for Dyck and semi-Dyck labeled path reachability // 2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON) / IEEE. — 2017. — P. 247–253.

[9] Ceri Stefano, Gottlob Georg, Tanca Letizia. What you Always Wanted to Know About Datalog (And Never Dared to Ask). // Knowledge and Data Engineering, IEEE Transactions on. — 1989. — 04. — Vol. 1. — P. 146 – 166.

[10] Chatterjee Krishnendu, Choudhary Bhavya, Pavlogiannis Andreas. Optimal Dyck Reachability for Data-Dependence and Alias Analysis // Proc. ACM Program. Lang. — 2017. — Dec. — Vol. 2, no. POPL. — Access mode: https://doi.org/10.1145/3158118.

[11] Chaudhary Anoop, Faisal Abdul. Role of graph databases in social networks. — 2016. — 06.

[12] CFL-reachability in subcubic time : Rep. / Technical report, IBM Research Report RC24126 ; Executor: Swarat Chaudhuri : 2006.

[13] Chaudhuri Swarat. Subcubic Algorithms for Recursive State Machines. — POPL '08. — New York, NY, USA : Association for Computing Machinery, 2008. — P. 159–169. — Access mode: https://doi.org/10.1145/1328438.1328460.

[14] Chistikov Dmitry, Majumdar Rupak, Schepper Philipp. Subcubic Certificates for CFL Reachability // arXiv preprint arXiv:2102.13095. — 2021.

[15] Chomsky Noam. On certain formal properties of grammars // Information and Control. — 1959. — Vol. 2, no. 2. — P. 137–167. — Access mode: https://www.sciencedirect.com/science/article/pii/S0019995859903626.

[16] 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.

[17] Deleage Jean-Luc, Pierre Laurent. The Rational Index of the Dyck Language $D_1'^*$ // Theor. Comput. Sci. — 1986. — Nov. — Vol. 47, no. 3. — P. 335–343.

[18] An Experimental Study of Context-Free Path Query Evaluation Methods / Jochem Kuijpers, George Fletcher, Nikolay Yakovets, Tobias Lindaaker. — SSDBM '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 121–132. — Access mode: https://doi.org/10.1145/3335783.3335791.

[19] Fast algorithms for Dyck-CFL-reachability with applications to alias analysis / Qirun Zhang, Michael R Lyu, Hao Yuan, Zhendong Su // Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. — 2013. — P. 435–446.

[20] Heintze N., McAllester D. On the cubic bottleneck in subtyping and flow analysis // Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science. — 1997. — P. 342–351.

[21] Hellings Jelle. Path Results for Context-free Grammar Queries on Graphs // CoRR. — 2015. — Vol. abs/1502.02242. — 1502.02242.

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

[23] Hopcroft John E., Ullman Jeffrey D. Set merging algorithms // SIAM Journal on Computing. — 1973. — Vol. 2, no. 4. — P. 294–303.

[24] Hopcroft John E, Ullman Jeffrey D. An introduction to automata theory, languages, and computation. — Upper Saddle River, NJ : Pearson, 1979.

[25] Ibaraki T., Katoh N. On-line computation of transitive closures of graphs // Information Processing Letters. — 1983. — Vol. 16, no. 2. — P. 95–97. — Access mode: https://www.sciencedirect.com/science/article/pii/0020019083900339.

[26] Impagliazzo Russell, Paturi Ramamohan. On the complexity of k-SAT // Journal of Computer and System Sciences. — 2001. — Vol. 62, no. 2. — P. 367–375.

[27] Kahlon Vineet. Boundedness vs. unboundedness of lock chains: Characterizing decidability of pairwise cfl-reachability for threads communicating via locks // 2009 24th Annual IEEE Symposium on Logic In Computer Science / IEEE. — 2009. — P. 27–36.

[28] Kodumal John, Aiken Alex. The Set Constraint/CFL Reachability Connection in Practice. — PLDI '04. — New York, NY, USA : Association for Computing Machinery, 2004. — P. 207–218. — Access mode: https://doi.org/10.1145/996841.996867.

[29] Li Yuanbo, Zhang Qirun, Reps Thomas. On the Complexity of Bidirected Interleaved Dyck-Reachability // Proc. ACM Program. Lang. — 2021. — Jan. — Vol. 5, no. POPL. — Access mode: https://doi.org/10.1145/3434340.

[30] Mathiasen Anders Alnor, Pavlogiannis Andreas. The Fine-Grained and Parallel Complexity of Andersen's Pointer Analysis // Proc. ACM Program. Lang. — 2021. — Jan. — Vol. 5, no. POPL. — Access mode: https://doi.org/10.1145/3434315.

[31] Medeiros Ciro M., Musicante Martin A., Costa Umberto S. Efficient Evaluation of Context-Free Path Queries for Graph Databases // Proceedings of the 33rd Annual ACM Symposium on Applied Computing. — SAC '18. — New York, NY, USA : Association for Computing Machinery, 2018. — P. 1230–1237. — Access mode: https://doi.org/10.1145/3167132.3167265.

[32] Melski David, Reps Thomas. Interconvertibility of a class of set constraints and context-free-language reachability // Theoretical Computer Science. — 2000. — Vol. 248, no. 1. — P. 29–98. — PEPM'97. Access mode: https://www.sciencedirect.com/science/article/pii/S0304397500000499.

[33] Mendelzon Alberto O., Wood Peter T. Finding Regular Simple Paths in Graph Databases // SIAM Journal on Computing. — 1995. — Vol. 24, no. 6. — P. 1235–1258. — https://doi.org/10.1137/S009753979122370X.

[34] Nondeterministic extensions of the strong exponential time hypothesis and consequences for non-reducibility / Marco L Carmosino, Jiawei Gao, Russell Impagliazzo et al. // Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science. — 2016. — P. 261–270.

[35] On economical construction of the transitive closure of an oriented graph / Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, IgorAleksandrovich Faradzhev // Doklady Akademii Nauk / Russian Academy of Sciences. — 1970.

[36] One Algorithm to Evaluate Them All: Unified Linear Algebra Based Approach to Evaluate Both Regular and Context-Free Path Queries / E. Shemetova, Rustam Azimov, Egor Orachev et al. // ArXiv. — 2021. — Vol. abs/2103.14688.

[37] RDF - Semantic Web Standards. — Accessed: 2020-05-15. Access mode: https://www.w3.org/RDF/.

[38] Rehof Jakob, Fähndrich Manuel. Type-Base Flow Analysis: From Polymorphic Subtyping to CFL-Reachability // Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '01. — New York, NY, USA : Association for Computing Machinery, 2001. — P. 54–66. — Access mode: https://doi.org/10.1145/360204.360208.

[39] Reps Thomas. Program analysis via graph reachability // Information and Software Technology. — 1998. — Vol. 40, no. 11. — P. 701–726. — Access mode: https://www.sciencedirect.com/science/article/pii/S0950584998000937.

[40] Reps Thomas. Undecidability of context-sensitive data-dependence analysis // ACM Transactions on Programming Languages and Systems (TOPLAS). — 2000. — Vol. 22, no. 1. — P. 162–186.

[41] SPARQL 1.1 overview. — Accessed: 2020-05-15. Access mode: https://www.w3.org/TR/sparql11-query/.

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

[43] Scott Elizabeth, Johnstone Adrian. GLL Parsing // Electronic Notes in Theoretical Computer Science. — 2010. — Vol. 253, no. 7. — P. 177–189. — Proceedings of the Ninth Workshop on Language Descriptions Tools and Applications (LDTA 2009). Access mode: https://www.sciencedirect.com/science/article/pii/S1571066110001209.

[44] Scott Elizabeth, Johnstone Adrian, Hussain Shamsa Sadaf. Tomita-style generalised LR parsers // Royal Holloway University of London. — 2000.

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

[46] Summary-based context-sensitive data-dependence analysis in presence of callbacks / Hao Tang, Xiaoyin Wang, Lingming Zhang et al. // Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — 2015. — P. 83–95.

[47] Valiant Leslie G. General context-free recognition in less than cubic time // Journal of computer and system sciences. — 1975. — Vol. 10, no. 2. — P. 308–315.

[48] Wikipedia contributors. Datalog — Wikipedia, The Free Encyclopedia. — 2021. — [Online; accessed 6-May-2021]. Access mode: https://en.wikipedia.org/w/index.php?title=Datalog&oldid=1015453010.

[49] Williams Virginia Vassilevska. On some fine-grained questions in algorithms and complexity // Proceedings of the ICM / World Scientific. — Vol. 3. — 2018. — P. 3431–3472.

[50] Yan Dacong, Xu Guoqing, Rountev Atanas. Demand-driven context-sensitive alias analysis for Java // Proceedings of the 2011 International Symposium on Software Testing and Analysis. — 2011. — P. 155–165.

[51] Yannakakis Mihalis. Graph-Theoretic Methods in Database Theory // Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems. — PODS '90. — New York, NY, USA : Association for Computing Machinery, 1990. — P. 230–242. — Access mode: https://doi.org/10.1145/298514.298576.

[52] Yannakakis Mihalis. Hierarchical state machines // IFIP International Conference on Theoretical Computer Science / Springer. — 2000. — P. 315–330.

[53] Yellin Daniel M. Speeding up Dynamic Transitive Closure for Bounded Degree Graphs // Acta Inf. — 1993. — Apr. — Vol. 30, no. 4. — P. 369–384. — Access mode: https://doi.org/10.1007/BF01209711.

[54] Younger Daniel H. Recognition and parsing of context-free languages in time n^3 // Information and Control. — 1967. — Vol. 10, no. 2. — P. 189–208. — Access mode: https://www.sciencedirect.com/science/article/pii/S001999586780007X.

[55] Yuan Hao, Eugster Patrick. An Efficient Algorithm for Solving the Dyck-CFL Reachability Problem on Trees // Programming Languages and Systems / Ed. by Giuseppe Castagna. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. — P. 175–189.

[56] Yuval Gideon et al. AN ALGORITHM FOR FINDING ALL SHORTEST PATHS USING $N^{2.81}$ INFINITE-PRECISION MULTIPLICATIONS. — 1976.

[57] Zarrinkalam Fattane, Kahani Mohsen, Paydar Samad. Using graph database for file recommendation in PAD social network // 7'th International Symposium on Telecommunications (IST'2014). — 2014. — P. 470–475.

[58] Zhang Qirun. Conditional Lower Bound for Inclusion-Based Points-to Analysis // arXiv preprint arXiv:2007.05569. — 2020.

[59] Zheng Xin, Rugina Radu. Demand-driven alias analysis for C // Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — 2008. — P. 197–208.