Оглавление

Введение

Актуальность

Постановка задачи и ключевые термины

Для формальной постановки задачи потребуется ввести некоторые вспомогательные определения.

Определение 0.1 (Помеченный граф)**.** Ориентированный помеченный граф (edge-labeled graph) (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 обычный мультиграф, каждому ребру которого сопоставлена метка из алфавита Σ.

Определение 0.2 (Контекстно-свободная грамматика)**.**

Определение 0.3 (Контекстно-свободный язык)**.** Язык, распознаваемый контекстно-свободной грамматикой

Теперь определим саму задачу.

Определение 0.4 (Задача поиска путей с контекстно-свободными ограничениями)**.** Входной граф: G

Входная грамматика: $\mathcal{G}$

РКА входной грамматики: $\mathcal{R}$

Теперь будут введены некоторые понятие (в основном, из теории формальных языков), которые встретятся далее по тексту работы.

Определение 0.5 (Язык Дика)**.** Языком Дика на k типах скобок (D_k) называют контекстно-свободный язык над алфавитом $\Sigma_k = \{\alpha_1, \overline{\alpha}_1, \dots \alpha_k, \overline{\alpha}_k\}$, состоящий из правильных скобочных последовательностей на k типах скобок (α_i соответствует открывающей скобке, $\overline{\alpha}_i$ — закрывающей).

Задачу CFPQ для языка Дика называют также задачу Диковой достижимости (Dyck-reachability).

Определение 0.6 (Двунаправленный граф)**.** Помеченный граф $G = \langle V, E, \Sigma_k \rangle$ называется двунаправленным (bidirected), если в нём для каждого ребра (u, v, α_i) найдётся противоположное ребро $(v, u, \overline{\alpha}_i)$ и наоборот.

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

Определение 0.7 (Система Непересекающихся Множеств (СНМ))**.**

Цель и задачи

Достигнутые результаты

Структура работы

1. Обзор литературы

1.1. Решения задачи в общем случае

1. $\mathcal{O}(n^3k^3)$ [11]

   Грамматика приводится к Нормальной форме Хомского, считается $dp_{i,j,c}$ — выводится ли путь $i \rightsquigarrow j$ из нетерминала s, при добавлении 1 ($dp_{i,j,A} = 1$) перебираются все соседние нетерминалы B, т.ч. $\exists C \to AB$ (или $C \to BA$) и $k \in V(G)$, и если $dp_{j,k,B}$ (или $dp_{k,i,B}$), то $dp_{i,k,C} = 1$ (или $dp_{k,j,C} = 1$) и (i, k, C) добавляется в рабочую очередь.

2. $\mathcal{O}(n^3k^3/\log n)$ [4]

   Алгоритм [11], к которому применили метод 4 русских

1.2. 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 и потому более быстрые решения. Этим уже занимались, сейчас мы выпишем всё, что на текущий момент известно:

1. Язык Дика $\mathcal{O}(n^3k)$ [9]

   Просто применить алгоритм Репса [11] и нормально оценить время работы.

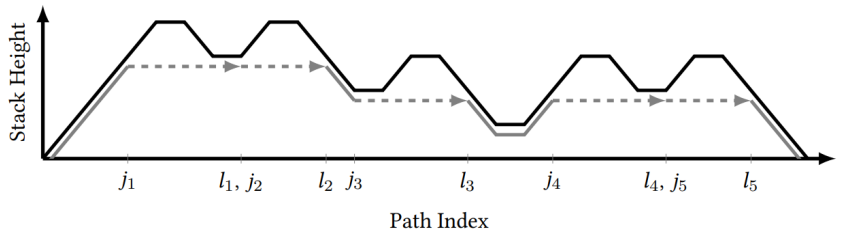
2. Язык Дика (почти) $\mathcal{O}(n^3)$ [12]

   Ещё более точный анализ алгоритм Репса [11], учитывающий, что построенный (для конкретного анализа) граф содержит константное число скобок

3. Язык Дика на одном типе скобок D_1 $\mathcal{O}(n^\omega \log^2 n)$ [10]

   Определение 1.1 (Bell-shaped путь)**.** Путь, который понимается строго вверх, потом сколько-то идёт ровно (ε-рёбра), потом спускается строго вниз

   Ищем bell-shaped пути: удваиваем рёбра, ищем пути с серединкой из bell-shaped пути поменьше (так $\log n^2$ раз).



Сжимаем bell-shaped пути в ε-рёбра. Снова ищем и снова сжимаем. После каждого сжимания мы убираем все локальные максимумы. Чем больше был максимум, тем длиннее ε-ребро. Хуже всего, когда все новые рёбра длины 2. В любом случае путь становится короче хотя бы в 2 раза, так что таких итераций потребуется не более $\log n^2$ (есть лемма, что найдётся путь длины не более $\mathcal{O}(n^2)$).

4. Двунаправленные графы и язык Дика

   Существует несколько частных решений для задачи Диковой достижимости на двунаправленных графах:

   - Деревья [16]

     $\mathcal{O}(n \log n \log k)$ — центроиды + внутри что-то идейное

   - Общий случай [3]

     Решение основано на двух фактах. Первый: в двунаправленном графе формируются компоненты Диковой достижимости. Второй: если есть две вершины u, v и компонента Диковой достижимости C, такие что $u \xrightarrow{\alpha_i} C$ и $v \xrightarrow{\alpha_i} C$, то u и v тоже лежат в одной компоненте Диковой достижимости.

     Пользуясь этими фактами, алгоритм с помощью CHM'а поддерживает компоненты Диковой достижимости и исходящие из них рёбра, чтобы быстро искать новые пары вершин, принадлежащих одной компоненте.

     Итоговая асимптотика алгоритма $\mathcal{O}(m + n\alpha(n))$.

   - Interleaved Dyck reachability

     Алгоритм за $\mathcal{O}(n^7)$ для $D_1 \bigodot D_1$ достижимости на bidirected графах: `https://helloqirun.github.io/papers/popl21_yuanbo.pdf`

     Было ещё про это (там, вроде, про один из языков сказали, что он bounded, поэтому можно пересекать с регулярным): `https://dl.acm.org/doi/pdf/10.1145/3296979.3192378`

5. Граф-цепочка $\mathcal{O}(n^\omega)$ [13]

   CFPQ на графе-цепочке — просто задача КС-распознавания (CF-recognition). А она решается за перемножение булевых матриц [13]

6. Ацикличный граф $\mathcal{O}(n^\omega)$ [14]

   Ацикличный граф — это почти бамбук (= цепочка), нужно только его потопсорить (и где-то ещё быть аккуратным, я не совсем помню сведение)

7. Bounded-stack RSM $\mathcal{O}(n^3k^3/\log^2)$ [4]

   RSM, который не уходит в рекурсию (т.е. есть из конца ребра $\xrightarrow{S}$ не достижимо никакое ребро $\xrightarrow{S}$)

Тут применяется какое-то более хитрое (я ещё не разбиралась) итеративное транзитивное замыкание (что-то с dfs'ом, а потом ещё 4 русских сверху, кажется)

8. Hierarchical FSM $\mathcal{O}(n^\omega k^\omega)$ [4]

   RSM, в котором боксы упорядочены (топсорт) и бокс с меньшим номером содержит рёбра только с вызовами боксов с большим номером. Задают регулярный язык, но размер FSM может быть экспоненциальным относительно размера RSM.

   Алгоритм идёт в порядке, обратном топсорту, и считает транзитивное замыкание внутри бокса, чтобы провести все рёбра, которые ему соответствуют.

2. Алгоритм, основанный на пересечении грамматик

2.1. Основные определения (пререквизиты)

Определение 2.1 (Конечный автомат (?))**.** НКА и ДКА

Определение 2.2 (Рекурсивный конечный автомат (РКА))**.** *Для простоты тут будет немного не такое определение, как в [1]*

Это набор компонент $M_1, M_2, \ldots, M_k$, где каждая компонента M_i — это пятёрка $\langle Q_i, \Sigma_i, En_i, Ex_1, \delta_i \rangle$, где

- 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
- $\delta_i \colon Q_i \times (\Sigma_i \cup \bigcup\limits_{j=1}^{k} En_i \times Ex_i) \to Q_i$ — функция перехода. У δ_i есть два типа переходов: *внутренние*, которые работаю 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 ДКА, на рёбрах которого могут быть "рекурсивные вызовы" других компонент.

TODO: картинка с примером

Определение 2.3 (Прямое произведение автоматов)**.**

Определение 2.4 (Транзитивное замыкание)**.**

Определение 2.5 (Инкрементальное транзитивное замыкание)**.**

2.2. Описание алгоритма

2.2.1. Алгоритм П

Я буду называть его алгоритм П

TODO: сделать на него ссылки везде

В данном разделе будет подробно описан алгоритм, предложенный в [5], в модификации которого будет состоять дальнейшая работа.

Главное идеей алгоритма является следующее замечание: любой помеченный граф можно рассматривать как НКА, в котором не обозначены начальное и конечные состояния. При этом, если зафиксировать конкретные вершины s и t как стартовое и конечное состояние, то полученный автомат будет задавать язык слов w, таких что существует путь из s в t, на котором читается w.

TODO: картинка с примером

Утверждение 2.1. *[7]*

Автомат A (НКА/ДКА), 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 ($\mathcal{L}(A) = \mathcal{L}(A_1) \cap \mathcal{L}(A_2)$)

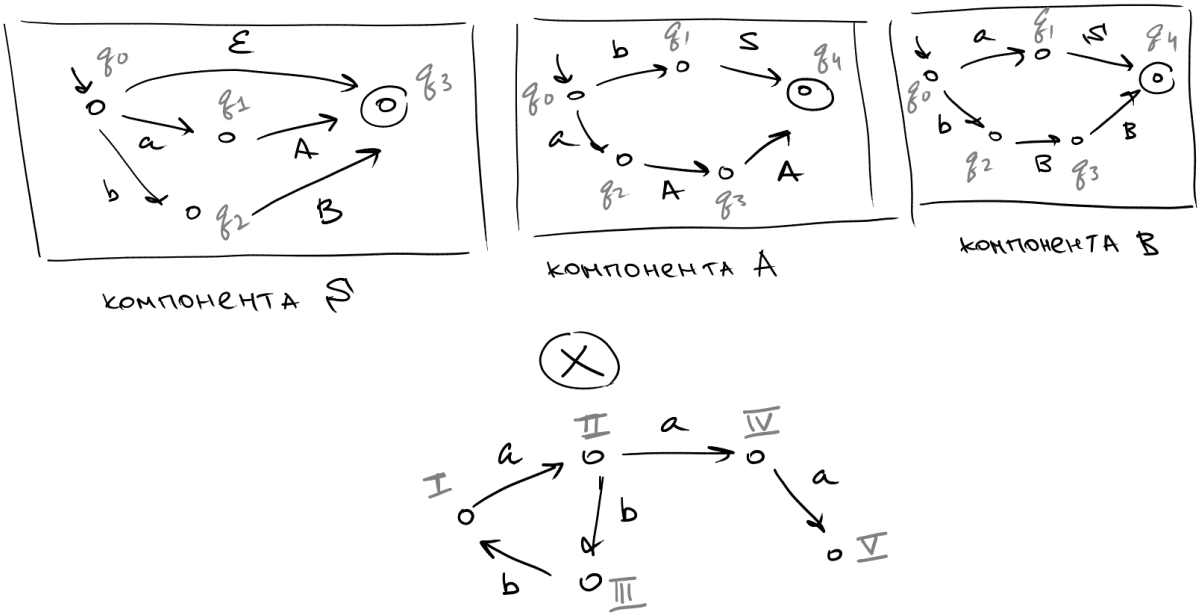
Данное утверждение остаётся верным, если один из языков задан РКА [2].

Пример 2.1 (Построение пересечения РКА и помеченного графа (НКА))**.** *РКА для грамматики, задающей язык слов, содержащих равное число букв a и b. Может быть задана следующими продукциями:*

$S \to \varepsilon \mid aA \mid bB$

$A \to bS \mid aAA$

$B \to aS \mid bBB$



TODO: *дорисовать пример (а потом перерисовать)*

Так что Алгоритм П можно описать так:

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 [14]. В случае же РКА задача осложняется наличием рекурсивных вызовов, которые разрешаются итеративно. (??)

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)
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)
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.

2.2.2. 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)

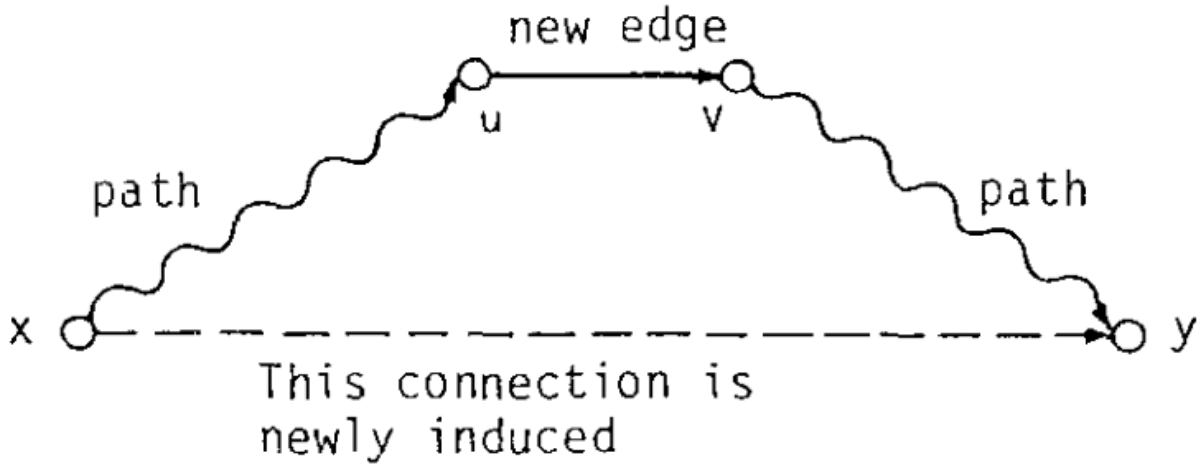
Listing 2 Алгоритм достижимости для РКА (2)

```
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 [8]. 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. Время работы

2.3.1. Алгоритм П

2.3.2. Алгоритм П2

2.4. Выводы и результаты по главе

3. Результаты (1)

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 [15] 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

- Неориентированные графы

  Вот тут получаем нетривиальные результаты, про них в следующем разделе

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.Join(u, v, Q)                        ▷ 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 ⟨стартовое состояние→конечное состояние⟩.

Опишем подробно структуру используемого СНМ (в листинге 4 приведён псевдокод функций *Join()* и *Get()*).

Listing 4 Система Непересекающихся Множеств

```
1: Structure DisjointSetUnion
2:     function DisjointSetUnion(V)
3:         for v ∈ V do
4:             P[v] ← v                               ▷ Предкок
5:             R[v] ← 0                               ▷ Ранг
6:             En[v] ← {v}                            ▷ Список стартовых вершин поддерева
7:             Ex[v] ← {v}                            ▷ Список конечных вершин поддерева
8:
```

TODO: Дописать код

За основу взята стандартная реализация [6], использующая подвешенные деревья.

TODO: *Надо ли её расписывать?*

Дополнительно в корнях хранятся списки всех начальных и конечных состояний компоненты. При добавлении ребра в операции *Join* перебираются все пары началь-

ная/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 $\mathcal{O}(n + m * \alpha(m^* + n, n))$*

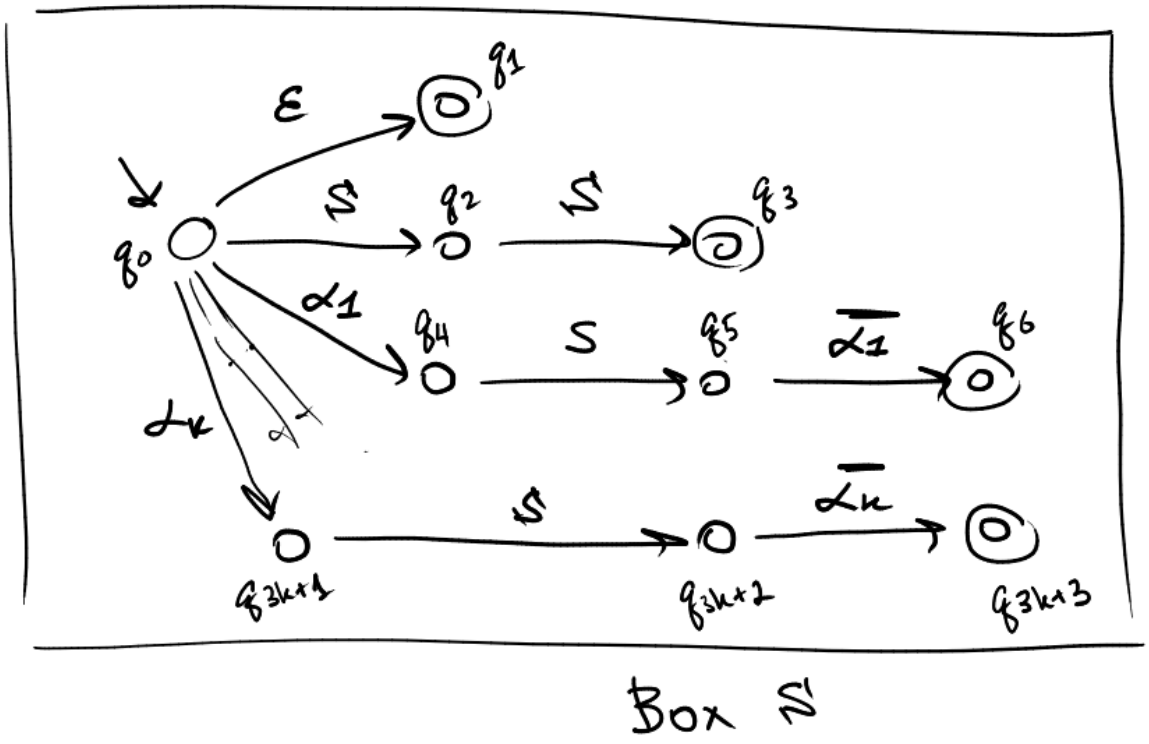
Доказательство.
TODO: *$m*$ джойнов, а проходы по спискам не долгие, так как каждый раз генерируем новое ребро* □

3.3. Корректность для неориентированных графов и некоторых классов грамматик

3.4. Корректность для двунаправленных графов и языка Дика

Замечание. For bidirected graphs the Dyck-reachability relation forms an equivalence, i.e., for all bidirected graphs G, for every pair of nodes um v, we have that v is Dyck-reachable from u iff u is Dyck-reachable from v.

Замечание. Dyck language RSM-grammar box (there is only one).



Теорема 3.2. *The Algorithm* **??** *works correctly on bidirected graphs and Dyck grammars.*

Доказательство. We only need to prove, that for every pair $u, v \in V(\mathcal{G})$ there is a path $(q_s, u) \rightsquigarrow (q_{f_i}, v)$ in (**directed**) Kronecker product $G \otimes \mathcal{G}$ iff there is such path $(q_s, u) \rightsquigarrow (q_{f_j}, v)$ in an **undirected** product (there q_s is the initial state and q_{f_i}, q_{f_j} are some final states of G respectively).

$\Rightarrow$

Obviously (if (q_f, v) is reachable from (q_s, u) by directed edges, it is all the more reachable by undirected edges).

$\Leftarrow$

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 3.4 $v \xrightarrow{S} u$ 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

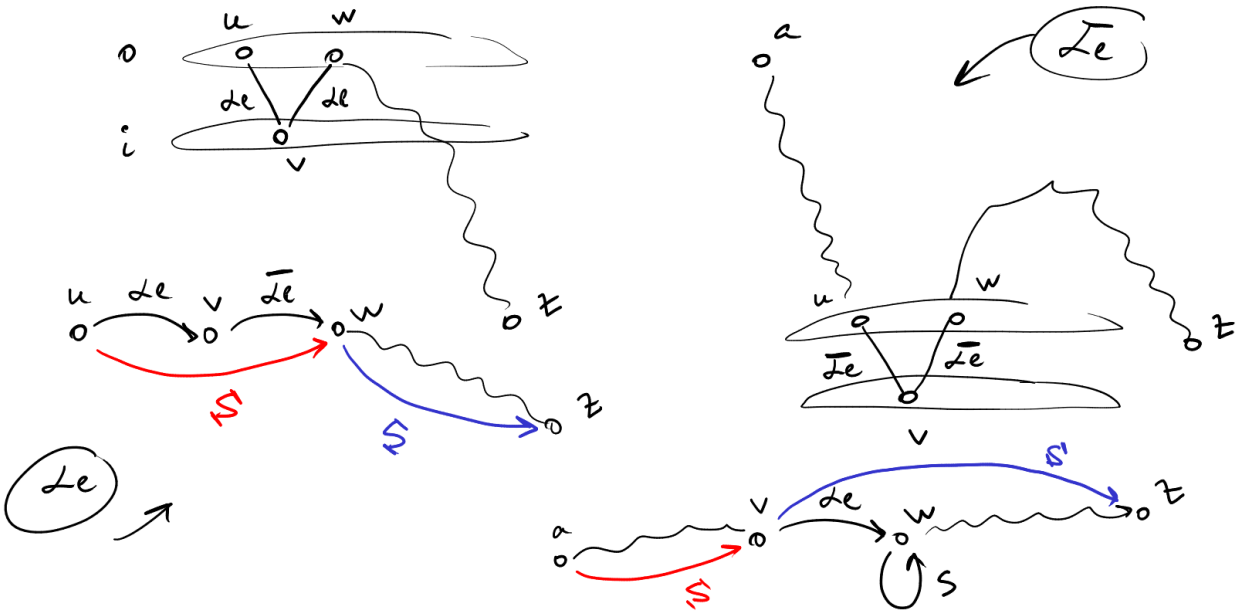
- $\overline{\alpha_l}$-label

  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.



□

4. Результаты (2)

4.1. Алгоритм для языка Дика на одном типе скобок

Список литературы

[1] 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.

[2] Beigel Richard, Gasarch William. A Proof that if L = L1 ⊠ L2 where L1 is CFL and L2 is Regular then L is Context Free Which Does Not use PDA's.

[3] 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.

[4] Chaudhuri Swarat. Subcubic Algorithms for Recursive State Machines // Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '08. — New York, NY, USA : Association for Computing Machinery, 2008. — P. 159–169. — Access mode: https://doi.org/10.1145/1328438.1328460.

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

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

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.