Приложения теории формальных языков и синтаксического анализа

Семён Григорьев

15 апреля 2020 г.

Оглавление

Список авторов

- **Семён Григорьев**
  Санкт-Петербургский государственный университет, Университетская набережная, 7/9, Санкт-Петербург, 199034, Россия
  s.v.grigoriev@spbu.ru

  JetBrains Research, Приморский проспект 68-70, здание 1, Санкт-Петербург, 197374, Россия
  semyon.grigorev@jetbrains.com

- **Екатерина Вербицкая**
  JetBrains Research, Приморский проспект 68-70, здание 1, Санкт-Петербург, 197374, Россия
  ekaterina.verbitskaya@jetbrains.com

- **Дмитрий Кутленков**
  Санкт-Петербургский государственный университет, Университетская набережная, 7/9, Санкт-Петербург, 199034, Россия
  kutlenkov.dmitri@gmail.com

Введение

Теория формальных языков находит применение не только для ставших уже классическими задач синтаксического анализа кода (языков программирования, искусственных языков) и естественных языков, но и в других областях, таких как статический анализ кода, графовые базы данных, биоинформатика, машинное обучение.

Напримемер, в машинном обучении использование формальных граммтик позволяет передать искусственной нейронной сети, предназначенной для генерации цепочек с определёнными свойствами (генеративной нейронной сети), знания о синтаксической структуре этих цепочек, что позволяет существенно упростить процесс обучения и повысить качество результата [46]. Вместе с этим, развиваются подходы, позволяющие нейронным сетям наоборот извлекать синтаксическую структуру (строить дерево вывода) для входных цепочек [41, 42].

В биоинформатике формальные граммтики нашли широкое применение для описания особенностей вторичной структуры геномных и белковых последовательностей [27, 5, 93]. Соответствующие алгоритмы синтаксического анализа используются при создании инструментов обработки данных.

Таким образом, теория формальных языков выступает в качестве основы для многих прикладных областей, а алгоритмы синтаксического анлиза применимы не только для обработки естественных языков или языков программирования. Нас же в данной работе будет интересовать применение теории формальных языков и алгоритмов синтаксического анализа для анализа графовым базам данных и для статического анализу кода.

Одна из классических задач, связанных с анализом графов — это поиск путей в графе. Возможны различные формулировки этой задачи. В некоторых случайх необходимо выяснить, существует ли путь с определёнными свойствами между двумя выбранными вершинами. В других же ситуациях необходимо найти все пути в графе, удовлетворяющие некоторым свойствам или ограничениям. Например, указать, что искомый путь должен быть простым, кратчайшим, гамильтоновым и так далее.

Один из способов задавать ограничения на пути в графе основан на использовании формальных языков. Базоваое определение языка говорит нам, что язык — это множество слов над некоторым алфавитом. Если рассмотреть граф, рёбра которого помечены символами из алфавита, то путь в таком графе будет задавать слово: достаточно соединить последовательно символы, лежащие на рёбрах пути. Множество же таких путей будет задавать множество слов или язык. Таким образом, если мы хотим найти некоторое множество путей в графе, то в качестве ограничения можно описать язык, который должно задавать это множество. Иными словами, задача поиска путей может быть сформулирована следующим образом: необходимо найти такие пути в графе, что слова, получаемые конкатенацей меток их рёбер, принадлежат заданному языку. Такой класс задач будем называть задачами поиска путей с ограничениям в теринах формальных языков.

Подобный класс задач часто возникает в областях, связанных с анализом граф-структурированных данных и активно исследуется [11, 6, 38, 84, 10, 11]. Исследуются как классы языков, применяемых для задания ограничений, так и различные постановки задачи.

Граф-структурированный данные встречаются не только в графовых базах данных, но и при статическом анализе кода: по программе можно построить различные графы отображающие её свойства. Скажем, граф вызовов, граф потока данны и так далее. Оказывается, что поиск путей в специального вида графах с сипользованием ограничений в терминах формальных языков позволяет исследовать некоторые свойства программы. Например проводить межпроцедурный анализ указателей или анализ алиасов [92, 88, 90], строить срезы программ [64], проводить анализ типов [61].

В данной работе пердставлен ряд алгоритмов для поиска путей с ограничениями в терминах формальных языков. Основной акцент булет сделан на контекстно-свободных языках, однако будут затронуты и другие классы: регулярные и конъюнктивные языки. Будет показано, что теория формальных языков и алгоритмы синатксического анализа применимы не только для анализа языков программирования или естественных языков, а также для анализа графовых баз данных и статического анализа кода, что приводит к возникновению новых задач и переосмыслению старых.

Структура данной работы такова. Сперва, в главе 1 мы рассмотрим основные понятия из теории графов, необходимые в данной работе. Затем, в главе 2 мы введём основные понятия из теории формальных языков. Далее, в главе 3 рассмотрим различные варианты постановки задачи поиска путей с ограничениями в терминах формальнх языков, обсудим базовые свойства задач, её разрешимость в различных постановках и т.д.. И в тоге зафикси-

руем постановку, которую будем изучать далее. После этого, в главах 4–11 мы будем подробно рассматривать различные алгоритмы решения этой задачи, попутно вводя специфичные для рассматриваемого алгоритма структуры данных. Большинство алгоритмов будут основаны на классических алгоритмах синтаксического анализа, таких как CYK или LR. Все гравы, начиная с 1, снабжены списком вопросов и задач для самостоятельного решения и закрепления материала.

Глава 1

Общие сведения теории графов

В данном разделе мы дадим определения базовым понятиям из теории графов, рассмотрим несколько классических задач из области анализа графов и алгоритмы их решения. Всё это понадобится нам при последующей работе.

1.1 Основные определения

Определение 1.1.1. *Граф* $\mathcal{G} = \langle V, E, L \rangle$, где V — конечное множество вершин, E — конечное множество рёбер, т.ч. $E \subseteq V \times L \times V$, L — конечное множество меток рёбер.

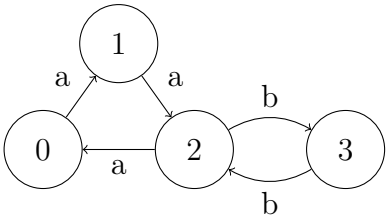
В дальнейшем нам будут нужны конечные ориентированные помеченные графы. Мы будем использовать термин *граф* подразумевая именно конечный ориентированный помеченный граф, если только не оговорено противное.

Также мы будем считать, что все вершины занумерованы подряд с нуля. То есть можно считать, что V — это отрезок неотрицательный целых чисел.

Пример 1.1.1 (Пример графа и его графического представления)**.** Пусть дан граф

$$\mathcal{G}_1 = \langle \{0, 1, 2, 3\}, \{(0, a, 1), (1, a, 2), (2, a, 0), (2, b, 3), (3, b, 2)\}, \{a, b\} \rangle.$$

Графическое представление графа $\mathcal{G}_1$:

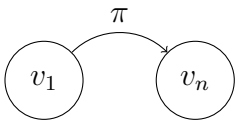


Определение 1.1.2. *Ребро* ориентированного помеченного графа $\mathcal{G} = \langle V, E, L \rangle$ это упорядоченная тройка $e = (v_i, l, v_j) \in V \times L \times V$.

Пример 1.1.2. $(0, a, 1)$ и $(3, b, 2)$ — это рёбра графа $\mathcal{G}_1$. При этом, $(3, b, 2)$ $(2, b, 3)$ — это разные рёбра, что видно из рисунка.

Определение 1.1.3. *Путём* π в графе $\mathcal{G}$ будем называть последовательность рёбер такую, что для любых двух последовательных рёбер $e_1 = (u_1, l_1, v_1)$ и $e_2 = (u_2, l_2, v_2)$ в этой последовательности, конечная вершина первого ребра является начальной вершиной второго, то есть $v_1 = u_2$. Будем обозначать путь из вершины v_0 в вершину v_n как

$$v_0 \pi v_n = e_0, e_1, \ldots, e_{n-1} = (v_0, l_0, v_1), (v_1, l_1, v_2), \ldots, (v_{n-1}, l_n, v_n).$$



Пример 1.1.3. $(0, a, 1)(1, a, 2) = 0\pi_1 2$ — путь из вершины 0 в вершину 2 в графе $\mathcal{G}_1$. При этом, $(0, a, 1)(1, a, 2)(2, b, 3)(3, b, 2) = 0\pi_2 2$ — это тоже путь из вершины 0 в вершину 2 в графе $\mathcal{G}_1$, но он не равен $0\pi_1 2$.

Нам потребуется также отношение, отражающее факт существования пути между двумя вершинами.

Определение 1.1.4. *Отношение достижимости* в графе: $(v_i, v_j) \in P \iff \exists v_i \pi v_j$.

Отметим, что рефлексивность этого отношения часто зависит от контекста. В некоторых задачах по-умолчанию $(v_i, v_i) \notin P$, а чтобы это было верно, требуется явное наличие ребра-петли.

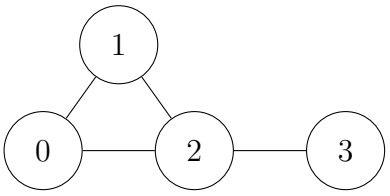
Один из способов задать граф — это задать его *матрицу смежности*.

Определение 1.1.5. *Матрица смежности* графа $\mathcal{G} = \langle V, E, L \rangle$ — это квадратная матрица M размера $n \times n$, где $|V| = n$ и ячейки которой содержат множества. При этом $l \in M[i, j] \iff \exists e = (i, l, j) \in E$.

Заметим, что наше определение матрицы смежности отличается от классического, в котором матрица отражает лишь факт наличия хотя бы одного ребра и, соответственно, является булевой. То есть $M[i, j] = 1 \iff \exists e = (i, \_, j) \in E$.

Также можно встретить матрицы смежности в ячейках которых всё же хранится некоторая информация, однако, в единственном экземпляре. То есть запрещены параллельные рёбра. Такой подход часто можно встретить в задачах о кратчайших путях: в этом случае в ячейке хранится расстояние между двумя вершинами. При этом, так как в качестве весов часто рассматривают произвольные (в том числе отрицательные) числа, то в задачах о кратчайших путях отдельно вводят значение “бесконечность” для обозначения ситуации, когда между двумя вершинами нет пути или его длина ещё не известна.

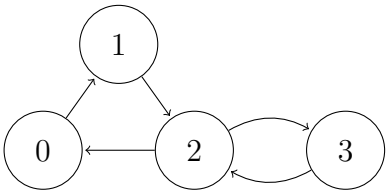
Пример 1.1.4. Неориентированный граф:



И его матрица смежности:

$$\begin{pmatrix} 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

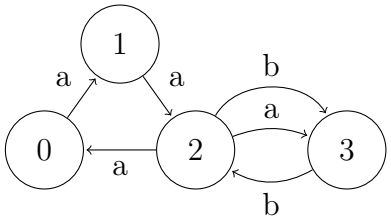
Пример 1.1.5. Ориентированный граф:



И его матрица смежности:

$$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

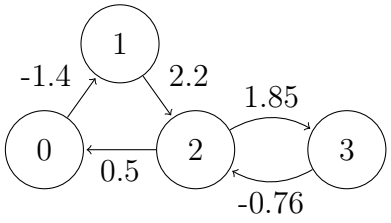
Пример 1.1.6. Помеченный граф:



И его матрица смежности:

$$\begin{pmatrix} \varnothing & \{a\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \{a\} & \varnothing \\ \{a\} & \varnothing & \varnothing & \{a,b\} \\ \varnothing & \varnothing & \{b\} & \varnothing \end{pmatrix}$$

Пример 1.1.7. Взвешенный граф для задачи о кратчайших путях:



И его матрица смежности (для задачи о кратчайших путях):

$$\begin{pmatrix} 0 & -1.4 & \infty & \infty \\ \infty & 0 & 2.2 & \infty \\ 0.5 & \infty & 0 & 1.85 \\ \infty & \infty & -0.76 & 0 \end{pmatrix}$$

Мы ввели лишь общие понятия. Специальные понятия, необходимые для изложения конкретного материала, будут даны в соответствующих главах.

1.2 Задачи поиска путей

Одна из классических задач анализа графов — это задача поиска путей между вершинами с различными ограничениями.

При этом, возможны различные постановки задачи. С одной стороны, по тому, что именно мы хотим получить в качестве результата:

- Наличие пути, удовлетворяющего ограничениям, в графе.

- Наличие пути, удовлетворяющего ограничениям, между некоторыми вершинами: задача достижимости. Достижима ли вершина v_1 из вершины v_2 по пути, удовлетворяющему ограничениям. Такая постановка требует лишь проверить существование, но не обязательно предоставлять путь.
- Поиск одного пути, удовлетворяющего ограничениям. Уже надо предъявлять путь.
- Поиск всех путей. Надо предоставить все пути.

С другой стороны, задачи различаются ещё и по тому, как фиксируем вершины:

- из одной вершины до всех
- между всеми парами вершин
- межу фиксированной парой вершин
- между двумя множествами вершин

Итого, можем сгенерировать прямое произведение различных постановок.

Ограничение, имеющее важное прикладное значение, — минимальность длины. Иными словами, важная прикладная задача — *поиск кратчайших путей в графе (англ. APSP — all-pairs shortest paths)*

Часто добавляют ограничения, что путь должен быть простым и другие.

1.3 APSP и транзитивное замыкание графа

Заметим, что отношение достижимости (1.1.4) является транзитивным. Действительно, если существует путь из v_i в v_j и путь из v_j в v_k, то существует путь из v_i в v_k.

Определение 1.3.1. *Транзитивным замыканием графа* называется транзитивное замыкание отношения достижимости по всему графу.

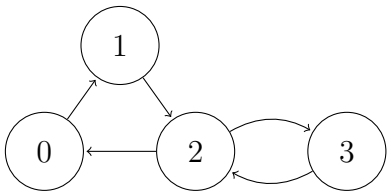
Как несложно заметить, транзитивное замыкание графа — это тоже граф. Более того, если решить задачу поиска кратчайших путей между всеми парами вершин, то мы построим транзитивное замыкание. Поэтому сразу рассмотрим алгоритм Флойда-Уоршелла [29, 65, 86], который является общим

Listing 1 Алгоритм Флойда-Уоршелла

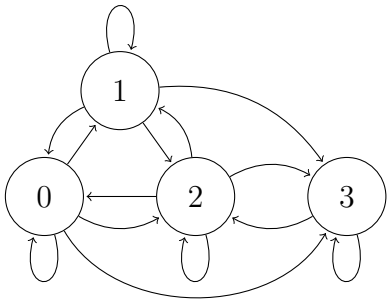
```
1: function FloydWarshall(𝒢)
2:     M ← матрица смежности 𝒢
3:     n ← |V(𝒢)|
4:     for k = 0; k < n; k++ do
5:         for i = 0; i < n; i++ do
6:             for j = 0; j < n; j++ do
7:                 M[i, j] ← min(M[i, j], M[i, k] + M[k, j])
8:     return M
```

алгоритмом поиска кратчайших путей (умеет обрабатывать рёбра с отрицательными весами, чего не может, например, алгоритм Дейкстры). Его сложность: $O(n^3)$, где n — количество вершин в графе. При этом, данный алгоритм легко упростить до алгоритма построения транзитивного замыкания.

Пример 1.3.1. Пусть дан следующий граф:



Построим его транзитивное замыкание:



Remark. На самом деле, петли у вершины 3 может и не быть, т.к. это зависит от определения.

Remark. Приведем список интересных работ по APSP для ориентированных графов с вещественными весами (здесь n — количество вершин в графе):

- M.L. Fredman (1976) — $O(n^3(\log\log n/\log n)^{\frac{1}{3}})$ — использование дерева решений [30]
- W. Dobosiewicz (1990) — $O(n^3/\sqrt{\log n})$ — использование операций на Word Random Access Machine [26]
- T. Takaoka (1992) — $O(n^3\sqrt{\log\log n/\log n})$ — использование таблицы поиска [76]
- Y. Han (2004) — $O(n^3(\log\log n/\log n)^{\frac{5}{7}})$ [33]
- T. Takoaka (2004) — $O(n^3(\log\log n)^2/\log n)$ [77]
- T. Takoaka (2005) — $O(n^3\log\log n/\log n)$ [78]
- U. Zwick (2004) — $O(n^3\sqrt{\log\log n}/\log n)$ [94]
- T.M. Chan (2006) — $O(n^3/\log n)$ — многомерный принцип "разделяй и властвуй" [20]
- и др.

1.4 APSP и произведение матриц

Алгоритм Флойда-Уоршелла можно переформулировать в терминах перемножения матриц. Для этого введём обозначение.

Определение 1.4.1. Пусть даны матрицы A и B размера $n \times n$. Определим операцию $\star$, которую называют *Min-plus matrix multiplication*:

$A \star B = C$ — матрица размера $n \times n$, т.ч. $C[i,j] = \min_{k=1\ldots n}\{A[i,k] + B[k,j]\}$

Также, обозначим за $D[i,j](k)$ минимальную длину пути из вершины i в j, содержащий максимум k рёбер. Таким образом, $D(1) = M$, где M — это матрица смежности, а решением APSP является $D(n-1)$, т.к. чтобы соединить n вершин требуется не больше $n-1$ рёбер.

$$\begin{gathered} D(1) = M \\ D(2) = D(1) \star M = M^2 \\ D(3) = D(2) \star M = M^3 \\ \ldots \\ D(n-1) = D(n-2) \star M = M^{n-1} \end{gathered}$$

Итак, мы можем решить APSP за $O(nK(n))$, где $K(n)$ — сложность алгоритма умножения матриц. Сразу заметим, что, например, $D(3)$ считать не обязательно, т.к. можем посчитать $D(4)$ как $D(2) \star D(2)$. Поэтому:

$$\begin{gathered} D(1) = M \\ D(2) = M^2 = M \star M \\ D(4) = M^4 = M^2 \star M^2 \\ D(8) = M^8 = M^4 \star M^4 \\ \cdots \\ D(2^{\log(n-1)}) = M^{2^{\log(n-1)}} = M^{2^{\log(n-1)}-1} \star M^{2^{\log(n-1)}-1} \\ D(n-1) = D(2^{\log(n-1)}) \end{gathered}$$

Теперь вместо n итераций нам нужно $\log n$. В итоге, сложность — $O(\log nK(n))$. Данный алгоритм называется *repeated squaring*[1].

1.5 Умножение матриц с субкубической сложностью

В предыдущем подразделе мы свели проблему APSP к проблеме min-plus matrix multiplication, поэтому взглянем на эффективные алгоритмы матричного умножения.

Сложность наивного произведения двух матриц составляет $O(n^3)$, это приемлемо только для малых матриц, для больших же лучше использовать алгоритмы с субкубической сложностью. Отметим, что мы называем сложность субкубической, если она равна $O(n^{3-\varepsilon})$, где $\varepsilon > 0$, иначе говоря, меньшей, чем $O(n^3)$.

Первый субкубический алгоритм опубликовал Ф. Штрассен в 1969 году, его сложность — $O(n^{\log_2 7}) \approx O(n^{2.81})$ [74]. Основная идея — рекурсивное разбиение на блоки 2×2 и вычисление их произведения с помощью только 7 умножений, а не 8. Впоследствии алгоритмы усовершенствовались до $\approx O(n^{2.5})$ [59, 14, 66, 22]. В настоящее время наиболее асимптотически быстрым является алгоритм Копперсмита — Винограда со сложностью $O(n^{2.376})$ [23].

Несмотря на то, что у приведенных алгоритмов неплохая алгоритмическая сложность, мы всё же не можем использовать их для вычисления min-plus matrix multiplication, так как в субкубических алгоритмах требуется, чтобы

[1]Пример решения APSP с помощью repeated squaring: `http://users.cecs.anu.edu.au/~Alistair.Rendell/Teaching/apac_comp3600/module4/all_pairs_shortest_paths.xhtml`

элементы образовывали кольцо. Покажем, что $\mathbb{R} \cup \{+\infty\}$ с операциями min и $+$ являются полукольцом, а не кольцом:

1. $min(a, b) = min(b, a)$
2. $min(min(a, b)), c) = min(a, min(b, c)))$
3. $min(a, +\infty) = min(+\infty, a) = a$, т.е. $+\infty$ — нейтральный элемент относительно операции min
4. $(a + b) + c = a + (b + c)$
5. $min(a, b) + c = min(a + c, b + c)$
6. $a + min(b, c) = min(a + b, a + c)$
7. $a + \infty = \infty + a = \infty$
8. Но для произвольного элемента a: $\nexists d$, т.ч. $min(a, d) = min(d, a) = +\infty$, т.е. для любого элемента нет обратного относительно операции min

Таким образом, вопрос о субкубических алгоритмах решения APSP всё ещё открыт [21]. Кроме того, более простая задача APSP с булевыми матрицами также не решена за субкубическую сложность. Приведем обоснование этого факта.

Определение 1.5.1. Матрица называется *булевой*, если она состоит из 0 и 1.

Булевы матрицы с поэлементными операциями дизъюнкции и конъюнкции являются полукольцом. Покажем это: пусть A, B и C — булевы матрицы, тогда:

1. $A \vee B = B \vee A$
2. $(A \vee B) \vee C = A \vee (B \vee C)$
3. $A \vee N = N \vee A = A$, где N — матрица, полностью состоящая из 0
4. $(A \wedge B) \wedge C = A \wedge (B \wedge C)$
5. $(A \vee B) \wedge C = (A \wedge C) \vee (B \wedge C)$
6. $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$
7. $A \wedge N = N \wedge A = N$

Булевы матрицы тоже не являются кольцом, т.к. не имеют обратный элемент относительно операции дизъюнкции (т.е. для произвольной булевой матрицы A: $\nexists D$, т.ч. D — булева матрица и $A \vee D = D \vee A = N$). Следовательно, субкубические алгоритмы не подходят для перемножения булевых матриц, т.к. в них используются обратные элементы (например, для разности). В частности, они не применимы к классической матрице смежности, которая ведёт себя как булева матрица.

1.6 Вопросы и задачи

1. Реализуйте абстракцию полукльца, позволяющую конструировать полукольца с произвольными операциями.

2. Реализуйте алгоритм произведения матриц над произвольным полукольцом. Используйте результат решения предыдущей задачи.

3. Используя результаты предыдущих задач, реализуйте алгоритм построения транзитивного замыкания через произведение матриц.

4. Используя результаты предыдущих задач, реализуйте алгоритм решения задачи APSP для ориентированного через произведение матриц.

5. Используя существующую библиотеку линейной алгебры для CPU, решите задачу построения транзитивного замыкания графа.

6. Используя существующую библиотеку линейной алгебры для CPU, решите задачу APSP для ориентированного графа.

7. Используя существующую библиотеку линейной алгебры для GPGPU, решите задачу построения транзитивного замыкания графа.

8. Используя существующую библиотеку линейной алгебры для GPGPU, решите задачу APSP для ориентированного графа.

9. Сравните произволительность решений предыдущих задач

Глава 2

Общие сведения теории формальных языков

В данной главе мы рассмотрим основные понятия из теории формальных языков, которые пригодятся нам в дальнейшем изложении.

Определение 2.0.1. *Алфавит* — это конечное множество. Элементы этого множества будем называть *символами*.

Пример 2.0.1. Примеры алфавитов

- Латинский алфавит $\Sigma = \{a, b, c, \ldots, z\}$
- Кириллический алфавит $\Sigma = \{$а, б, в, ..., я$\}$
- Алфавит чисел в шестнадцатеричной записи

$$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F\}$$

Традиционное обозначение для алфавита — Σ. Также мы будем использовать различные прописные буквы латинского алфавита. Для обозначения символов алфавита будем использовать строчные буквы латинского алфавита: $a, b, \ldots, x, y, z$.

Будем считать, что над алфавитом Σ всегда определена операция конкатенации $(\cdot) : \Sigma^* \times \Sigma^* \to \Sigma^*$. При записи выражений символ точки (обозначение операции конкатенации) часто будем опускать: $a \cdot b = ab$.

Определение 2.0.2. *Слово* над алфавитом Σ — это конечная конкатенация символов алфавита Σ: $\omega = a_0 \cdot a_1 \cdot \ldots \cdot a_m$, где ω — слово, а для любого i $a_i \in \Sigma$.

Определение 2.0.3. Пусть $\omega = a_0 \cdot a_1 \cdot \ldots \cdot a_m$ — слово над алфавитом Σ. Будем называть $m + 1$ *длиной слова* и обозначать как $|\omega|$.

Определение 2.0.4. *Язык* над алфавитом Σ — это множество слов над алфавитом Σ.

Пример 2.0.2. Примеры языков.

- Язык целых чисел в двоичной записи $\{0, 1, -1, 10, 11, -10, -11, \ldots\}$.
- Язык всех правильных скобочных последовательностей

$$\{(), (()), ()(), (())(), \ldots\}.$$

Любой язык над алфавитом Σ является подмножеством Σ^* — множества всех слов над алфавитом Σ

Заметим, что язык не обязан быть конечным множеством, в то время как алфавит всегда конечен и изучаем мы конечные слова.

Способы задания языков

- Перечислить все элементы. Такой способ работает только для конечных языков. Перечислить бесконечное множество не получится.
- Задать генератор — процедуру, которая возвращает очередное слово языка.
- Задать распознователь — процедуру, которая по данному слову может определить, принадлежит оно заданному языку или нет.

2.1 Контекстно-свободные грамматики и языки

Из всего многообразия нас будут интересовать прежде всего контекстно-свободные грамматики.

Определение 2.1.1. *Контекстно-свободная грамматика* — это четвёрка вида $\langle \Sigma, N, P, S \rangle$, где

- Σ — это терминальный алфавит;
- N — это нетерминальный алфавит;

- P — это множество правил или продукций, таких что каждая продукция имеет вид $N_i \to \alpha$, где $N_i \in N$ и $\alpha \in \{\Sigma \cup N\}^* \cup \varepsilon$;
- S — стартовый нетерминал. Отметим, что $\Sigma \cap N = \varnothing$.

Пример 2.1.1. Грамматика, задающая язык целых чисел в двоичной записи без лидирующих нулей: $G = \langle\{0, 1, -\}, \{S, N, A\}, P, S\rangle$, где P определено следующим образом:

$$\begin{array}{rcl} S & \to & 0 \mid N \mid -N \\ N & \to & 1A \\ A & \to & 0A \mid 1A \mid \varepsilon \end{array}$$

При спецификации грамматики часто опускают множество терминалов и нетерминалов, оставляя только множество правил. При этом нетерминалы часто обозначаются прописными латинскими буквами, терминалы — строчными, а стартовый нетерминал обозначается буквой S. Мы будем следовать этим обозначениям, если не указано иное.

Определение 2.1.2. *Отношение непосредственной выводимости.* Мы говорим, что 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

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

Определение 2.1.3. *Рефлексивно-транзитивное замыкание отношения* — это наименьшее рефлексивное и транзитивное отношение, содержащее исходное.

Определение 2.1.4. *Отношение выводимости* является рефлексивно-транзитивным замыканием отношения непосредственной выводимости

- $\alpha \overset{*}{\Rightarrow} \beta$ означает $\exists\gamma_0, \ldots \gamma_k : \ \alpha \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \ldots \Rightarrow \gamma_{k-1} \Rightarrow \gamma_k \Rightarrow \beta$
- Транзитивность: $\forall\alpha, \beta, \gamma \in \{\Sigma \cup N\}^* \cup \varepsilon : \ \alpha \overset{*}{\Rightarrow} \beta, \beta \overset{*}{\Rightarrow} \gamma \Rightarrow \alpha \overset{*}{\Rightarrow} \gamma$
- Рефлексивность: $\forall\alpha \in \{\Sigma \cup N\}^* \cup \varepsilon : \ \alpha \overset{*}{\Rightarrow} \alpha$
- $\alpha \overset{*}{\Rightarrow} \beta$ — α выводится из β
- $\alpha \overset{k}{\Rightarrow} \beta$ — α выводится из β за k шагов

- $\alpha \overset{+}{\Rightarrow} \beta$ — при выводе использовалось хотя бы одно правило грамматики

Пример 2.1.2. Пример вывода цепочки -1101 в грамматике:

$$\begin{array}{rcl} S & \to & 0 \mid N \mid -N \\ N & \to & 1A \\ A & \to & 0A \mid 1A \mid \varepsilon \end{array}$$

$$S \Rightarrow -N \Rightarrow -1A \Rightarrow -11A \overset{*}{\Rightarrow} -1101A \Rightarrow -1101$$

Определение 2.1.5 (Вывод слова в грамматике)**.** Слово $\omega \in \Sigma^*$ *выводимо в грамматике* $\langle \Sigma, N, P, S \rangle$, если существует некоторый вывод этого слова из начального нетерминала $S \overset{*}{\Rightarrow} \omega$.

Определение 2.1.6. *Левосторонний вывод.* На каждом шаге вывода заменяется самый левый нетерминал.

Пример 2.1.3. Пример левостороннего вывода цепочки в грамматике

$$\begin{array}{rcl} S & \to & AA \mid s \\ A & \to & AA \mid Bb \mid a \\ B & \to & c \mid d \end{array}$$

$$\boldsymbol{S} \Rightarrow \boldsymbol{A}A \Rightarrow \boldsymbol{B}bA \Rightarrow cb\boldsymbol{A} \Rightarrow cb\boldsymbol{A}A \Rightarrow cba\boldsymbol{A} \Rightarrow cbaa$$

Аналогично можно определить правосторонний вывод.

Определение 2.1.7. *Язык, задаваемый грамматикой* — множество строк, выводимых в грамматике $L(G) = \{\omega \in \Sigma^* \mid S \overset{*}{\Rightarrow} \omega\}$.

Определение 2.1.8. Грамматики G_1 и G_2 называются *эквивалентными*, если они задают один и тот же язык: $L(G_1) = L(G_2)$

Пример 2.1.4. Пример эквивалентных грамматик для языка целых чисел в двоичной системе счисления.

$$\begin{array}{rcl} \Sigma & = & \{0, 1, -\} \\ N & = & \{S, N, A\} \\ \\ S & \to & 0 \mid N \mid -N \\ N & \to & 1A \\ A & \to & 0A \mid 1A \mid \varepsilon \end{array}$$

$$\begin{array}{rcl} \Sigma & = & \{0, 1, -\} \\ N & = & \{S, A\} \\ \\ S & \to & 0 \mid 1A \mid -1A \\ A & \to & 0A \mid 1A \mid \varepsilon \end{array}$$

Определение 2.1.9. *Неоднозначная грамматика* — грамматика, в которой существует 2 и более левосторонних (правосторонних) выводов для одного слова.

Пример 2.1.5. Неоднозначная грамматика для правильных скобочных последовательностей

$$S \to (S) \mid SS \mid \varepsilon$$

Определение 2.1.10. *Однозначная грамматика* — грамматика, в которой существует не более одного левостороннего (правостороннего) вывода для каждого слова.

Пример 2.1.6. Однозначная грамматика для правильных скобочных последовательностей

$$S \to (S)S \mid \varepsilon$$

Определение 2.1.11. *Существенно неоднозначные языки* — языки, для которых невозможно построить однозначную грамматику.

Пример 2.1.7. Пример существенно неоднозначного языка

$$\{a^n b^n c^m \mid n, m \in \mathbb{Z}\} \cup \{a^n b^m c^m \mid n, m \in \mathbb{Z}\}$$

2.2 Дерево вывода

В некоторых случаях не достаточно знать порядок применения правил. Необходимо структурное представление вывода цепочки в грамматике. Таким представлением является *дерево вывода*.

Определение 2.2.1. Деревом вывода цепочки ω в грамматике $G = \langle \Sigma, N, S, P \rangle$ называется дерево, удовлетворяющее следующим свойствам.

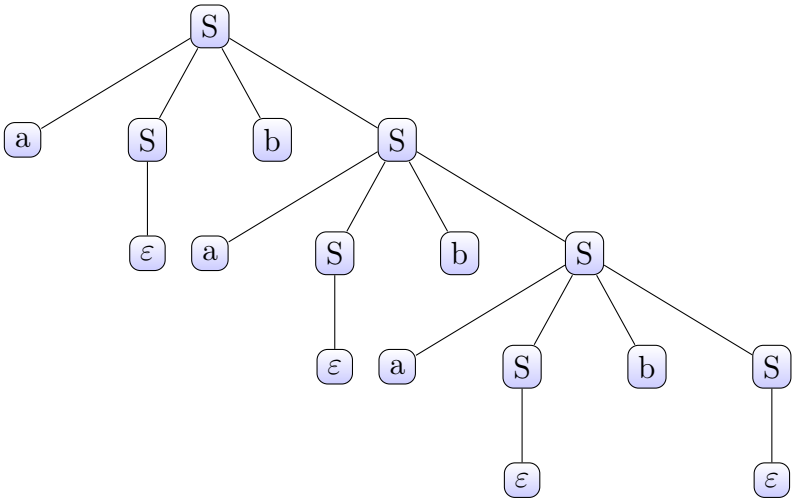
1. Помеченное: метка каждого внутреннего узла — нетерминал, метка каждого листа — терминал или ε.
2. Корневое: корень помечен стартовым нетерминалом.
3. Упорядоченное.

4. В дереве может существовать узел с меткой N_i и сыновьями $M_j \dots M_k$ только тогда, когда в грамматике есть правило вида $N_i \to M_j \dots M_k$.

5. Крона образует исходную цепочку ω.

Пример 2.2.1. Построим дерево вывода цепочки *ababab* в грамматике

$$G = \langle \{a, b\}, \{S\}, S, \{S \to a\ S\ b\ S, S \to \varepsilon\} \rangle$$



Теорема 2.2.1. Пусть $G = \langle \Sigma, N, P, S \rangle$ — КС-грамматика. Вывод $S \stackrel{*}{\Rightarrow} \alpha$, где $\alpha \in (N \cup \Sigma)^*, \alpha \neq \varepsilon$ существует $\Leftrightarrow$ существует дерево вывода в грамматике G с кроной α.

2.3 Пустота КС-языка

Теорема 2.3.1. Существует алгоритм, определяющий, является ли язык, порождаемый КС грамматикой, пустым.

Доказательство. Следующая лемма утверждает, что если в КС языке есть выводимое слово, то существует другое выводимое слово с деревом вывода не глубже количества нетерминалов грамматики. Для доказательства теоремы достаточно привести алгоритм, последовательно строящий все деревья глубины не больше количества нетерминалов грамматики, и проверяющий, являются ли такие деревья деревьями вывода. Если в результате работы алгоритма не удалось построить ни одного дерева, то грамматика порождает пустой язык. □

Лемма 2.3.2. Если в данной грамматике выводится некоторая цепочка, то существует цепочка, дерево вывода которой не содержит ветвей длиннее m, где m — количество нетерминалов грамматики.

Доказательство. Рассмотрим дерево вывода цепочки ω. Если в нем есть 2 узла, соответствующих одному нетерминалу A, обозначим их n_1 и n_2.

Предположим, n_1 расположен ближе к корню дерева, чем n_2.

$S \overset{*}{\Rightarrow} \alpha A_{n_1} \beta \overset{*}{\Rightarrow} \alpha\omega_1\beta; S \overset{*}{\Rightarrow} \alpha\gamma A_{n_2}\delta\beta \overset{*}{\Rightarrow} \alpha\gamma\omega_2\delta\beta$, при этом ω_2 является подцепочкой ω_1.

Заменим в изначальном дереве узел n_1 на n_2. Полученное дерево является деревом вывода $\alpha\omega_2\delta$.

Повторяем процесс замены одинаковых нетерминалов до тех пор, пока в дереве не останутся только уникальные нетерминалы.

В полученном дереве не может быть ветвей длины большей, чем m.

По построению оно является деревом вывода. □

2.4 Нормальная форма Хомского

Определение 2.4.1. Контекстно-свободная грамматика $\langle \Sigma, N, P, S \rangle$ находится в *Нормальной Форме Хомского*, если она содержит только правила следующего вида:

- $A \to BC$, где $A, B, C \in N$, S не содержится в правой части правила
- $A \to a$, где $A \in N, a \in \Sigma$
- $S \to \varepsilon$

Теорема 2.4.1. Любую КС грамматику можно преобразовать в НФХ.

Доказательство. Алгоритм преобразования в НФХ состоит из следующих шагов:

- Замена неодиночных терминалов
- Удаление длинных правил
- Удаление ε-правил

- Удаление цепных правил
- Удаление бесполезных нетерминалов

То, что каждый из этих шагов преобразует грамматику к эквивалентной, при этом является алгоритмом, доказано в следующих леммах. □

Лемма 2.4.2. Для любой КС-грамматики можно построить эквивалентную, которая не содержит правила с неодиночными терминалами.

Доказательство. Каждое правило $A \to B_0 B_1 \dots B_k, k \geq 1$ заменить на множество правил:

- $A \to C_0 C_1 \dots C_k$
- $\{C_i \to B_i \mid B_i \in \Sigma, C_i$ — новый нетерминал$\}$

□

Лемма 2.4.3. Для любой КС-грамматики можно построить эквивалентную, которая не содержит правил длины больше 2.

Доказательство. Каждое правило $A \to B_0 B_1 \dots B_k, k \geq 2$ заменить на множество правил:

- $A \to B_0 C_0$
- $C_0 \to B_1 C_1$
- $\dots$
- $C_{k-3} \to B_{k-2} C_{k-2}$
- $C_{k-2} \to B_{k-1} B_k$

□

Лемма 2.4.4. Для любой КС-грамматики можно построить эквивалентную, не содержащую ε-правил.

Доказательство. Определим ε-правила:

- $A \to \varepsilon$

- $A \to B_0 \dots B_k, \forall i: \ B_i$ — ε-правило.

Каждое правило $A \to B_0 B_1 \dots B_k$ заменяем на множество правил, где каждое ε-правило удалено во всех возможных комбинациях. □

Лемма 2.4.5. Можно удалить все цепные правила

Доказательство. Цепное правило — правило вида $A \to B$, где $A, B \in N$. *Цепная пара* — упорядоченная пара (A, B), в которой $A \overset{*}{\Rightarrow} B$, используя только цепные правила.

Алгоритм:

1. Найти все цепные пары в грамматике G. Найти все цепные пары можно по индукции: Базис: (A, A) — цепная пара для любого нетерминала, так как $A \overset{*}{\Rightarrow} A$ за ноль шагов. Индукция: Если пара (A, B_0) — цепная, и есть правило $B_0 \to B_1$, то (A, B_1) — цепная пара.
2. Для каждой цепной пары (A, B) добавить в грамматику G' все правила вида $A \to a$, где $B \to a$ — нецепное правило из G.
3. Удалить все цепные правила

Пусть G — контекстно-свободная грамматика. G' — грамматика, полученная в результате применения алгоритма к G. Тогда $L(G) = L(G')$. □

Определение 2.4.2. Нетерминал A называется *порождающим*, если из него может быть выведена конечная терминальная цепочка. Иначе он называется *непорождающим*.

Лемма 2.4.6. Можно удалить все бесполезные (непорождающие) нетерминалы

Доказательство. После удаления из грамматики правил, содержащих непорождающие нетерминалы, язык не изменится, так как непорождающие нетерминалы по определению не могли участвовать в выводе какого-либо слова.

Алгоритм нахождения порождающих нетерминалов:

1. Множество порождающих нетерминалов пустое.
2. Найти правила, не содержащие нетерминалов в правых частях и добавить нетерминалы, встречающихся в левых частях таких правил, в множество.

3. Если найдено такое правило, что все нетерминалы, стоящие в его правой части, уже входят в множество, то добавить в множество нетерминалы, стоящие в его левой части.

4. Повторить предыдущий шаг, если множество порождающих нетерминалов изменилось.

В результате получаем множество всех порождающих нетерминалов грамматики, а все нетерминалы, не попавшие в него, являются непорождающими. Их можно удалить. □

Пример 2.4.1. Приведем в Нормальную Форму Хомского однозначную грамматику правильных скобочных последовательностей: $S \to aSbS \mid \varepsilon$

Первым шагом добавим новый нетерминал и сделаем его стартовым:

$$\begin{aligned} S_0 &\to S \\ S &\to aSbS \mid \varepsilon \end{aligned}$$

Заменим все терминалы на новые нетерминалы:

$$\begin{aligned} S_0 &\to S \\ S &\to LSRS \mid \varepsilon \\ L &\to a \\ R &\to b \end{aligned}$$

Избавимся от длинных правил:

$$\begin{aligned} S_0 &\to S \\ S &\to LS' \mid \varepsilon \\ S' &\to SS'' \\ S'' &\to RS \\ L &\to a \\ R &\to b \end{aligned}$$

Избавимся от ε-продукций:

$$
\begin{aligned}
S_0 &\to S \mid \varepsilon \\
S &\to LS' \\
S' &\to S'' \mid SS'' \\
S'' &\to R \mid RS \\
L &\to a \\
R &\to b
\end{aligned}
$$

Избавимся от цепных правил:

$$
\begin{aligned}
S_0 &\to LS' \mid \varepsilon \\
S &\to LS' \\
S' &\to b \mid RS \mid SS'' \\
S'' &\to b \mid RS \\
L &\to a \\
R &\to b
\end{aligned}
$$

Определение 2.4.3. Контекстно-свободная грамматика $\langle \Sigma, N, P, S \rangle$ находится в *ослабленной Нормальной Форме Хомского*, если она содержит только правила следующего вида:

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

То есть ослабленная НФХ отличается от НФХ тем, что:

1. ε может выводиться из любого нетерминала
2. S может появляться в правых частях правил

2.5 Лемма о накачке

Лемма 2.5.1. Пусть L — контекстно-свободный язык над алфавитом Σ, тогда существует такое n, что для любого слова $\omega \in L$, $|\omega| \geq n$ найдутся слова $u, v, x, y, z \in \Sigma^*$, для которых верно: $uvxyz = \omega, vy \neq \varepsilon, |vxy| \leq n$ и для любого $k \geq 0$ $uv^k xy^k z \in L$.

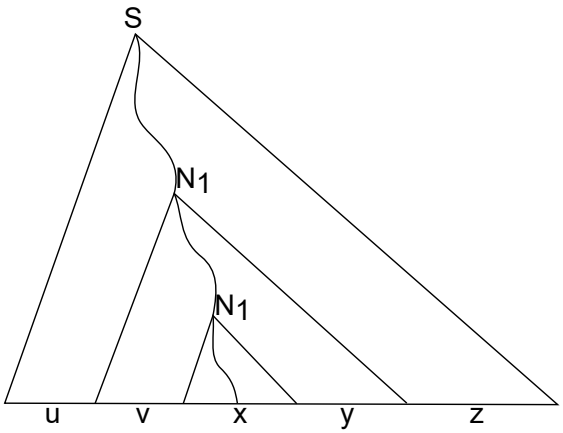


Рис. 2.1: Разбиение цепочки для леммы о накачке

Идея доказательства леммы о накачке.

1. Для любого КС языка можно найти грамматику в нормальной форме Хомского.

2. Очевидно, что если брать достаточно длинные цепочки, то в дереве вывода этих цепочек, на пути от корня к какому-то листу обязательно будет нетерминал, встречающийся минимум два раза. Если m — количество нетерминалов в НФХ, то длины 2^{m+1} должно хватить. Это и будет n из леммы.

3. Возьмём путь, на котором есть хотя бы дважды повторяется некоторый нетерминал. Скажем, это нетерминал N_1. Пойдём от листа по этому пути. Найдём первое появление N_1. Цепочка, задаваемая поддеревом для этого узла — это x из леммы.

4. Пойдём дальше и найдём второе появление N_1. Цепочка, задаваемая поддеревом для этого узла — это vxy из леммы.

5. Теперь мы можем копировать кусок дерева между этими повторениями N_1 и таким образом накачивать исходную цепочку.

Надо только проверить выполение ограничений на длины.

Нахождение разбиения и пример накачки продемонстрированы на рисунках 2.1 и 2.2.

Для примера предлагается проверить неконтекстно-свободность языка $L = \{a^n b^n c^n \mid n > 0\}$.

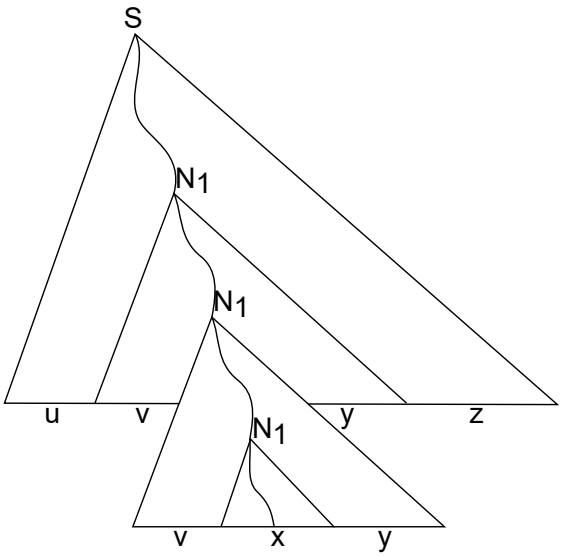


Рис. 2.2: Пример накачки цепочки с рисунка 2.1

2.6 Замкнутость КС языков относительно операций

Теорема 2.6.1. Контекстно-свободные языки замкнуты относительно следующих операций:

1. Объединение: если L_1 и L_2 — контекстно-свободные языки, то и $L_3 = L_1 \cup L_2$ — контекстно-свободный.

2. Конкатенация: если L_1 и L_2 — контекстно-свободные языки, то и $L_3 = L_1 \cdot L_2$ — контекстно-свободный.

3. Замыкание Клини: если L_1 — контекстно-свободный, то и $L_2 = \bigcup\limits_{i=0}^{\infty} L_1^i$ — контекстно-свободный.

4. Разворот: если L_1 — контекстно-свободный, то и $L_2 = {L_1}^r$ — контекстно-свободный.

5. Пересечение с регулярными языками: если L_1 — контекстно-свободный, а L_2 — регулярный, то $L_3 = L_1 \cap L_2$ — контекстно-свободный.

6. Разность с регулярными языками: если L_1 — контекстно-свободный, а L_2 — регулярный, то $L_3 = L_1 \setminus L_2$ — контекстно-свободный.

Для доказательства пунктов 1–4 можно построить КС граммтику нового языка имея грамматики для исходных. Будем предполагать, что множества нетерминальных символов различных граммтик для исходных языков не пересекаются.

1. $G_1 = \langle \Sigma_1, N_1, P_1, S_1 \rangle$ — граммтика для L_1, $G_1 = \langle \Sigma_2, N_2, P_2, S_2 \rangle$ — граммтика для L_2, тогда $G_3 = \langle \Sigma_1 \cup \Sigma_2, N_1 \cup N_2 \cup \{S_3\}, P_1 \cup P_2 \cup \{S_3 \to S_1 \mid S_2\}, S_3 \rangle$ — граммтика для L_3.

2. $G_1 = \langle \Sigma_1, N_1, P_1, S_1 \rangle$ — граммтика для L_1, $G_1 = \langle \Sigma_2, N_2, P_2, S_2 \rangle$ — граммтика для L_2, тогда $G_3 = \langle \Sigma_1 \cup \Sigma_2, N_1 \cup N_2 \cup \{S_3\}, P_1 \cup P_2 \cup \{S_3 \to S_1 S_2\}, S_3 \rangle$ — граммтика для L_3.

3. $G_1 = \langle \Sigma_1, N_1, P_1, S_1 \rangle$ — граммтика для L_1, тогда $G_2 = \langle \Sigma_1, N_1 \cup \{S_2\}, P_1 \cup \{S_2 \to S_1 S_2 \mid \varepsilon\}, S_2 \rangle$ — граммтика для L_2.

4. $G_1 = \langle \Sigma_1, N_1, P_1, S_1 \rangle$ — граммтика для L_1, тогда $G_2 = \langle \Sigma_1, N_1, \{N^i \to \omega^R \mid N^i \to \omega \in P_1\}, S_1 \rangle$ — граммтика для L_2.

Чтобы доказать замкнутость относительно пересечения с регулярными языками, построим по КС грамматике рекурсивный автомат R_1, по регулярному выражению — детерминированный конечный автомат R_2, и построим их прямое произведение R_3. Переходы по терминальным символам в новом автомате возможны тогда и только тогда, когда они возможны одновременно и в исходном рекурсивном автомате и в исходном конечном. За рекурсивные вызовы отвечает исходныа рекурсивный автомат. Значит цепочка принимается R_3 тогда и только тогда, когда она принимается одновременно R_1 и R_2: так как состояния R_3 — это пары из состояния R_1 и R_2, то по трассе вычислений R_3 мы всегда можем построить трассу для R_1 и R_2 и наоборот.

Чтобы доказать замкнутость относительно разности с регулятным языком, достаточно вспомнить, что регулярные языки замкнуты относительно дополнения, и выразить разность через пересечение с дополнением:

$$L_1 \setminus L_2 = L_1 \cap \overline{L_2}$$

□

Теорема 2.6.2. Контекстно-свободные языки не замкнуты относительно следующих операций:

1. Пересечение: если L_1 и L_2 — контекстно-свободные языки, то и $L_3 = L_1 \cap L_2$ — не контекстно-свободный.

2. Разность: если L_1 и L_2 — контекстно-свободные языки, то и $L_3 = L_1 \setminus L_2$ — не контекстно-свободный.

Чтобы доказать незамкнутость относительно пресечения, рассмотрим языки $L_1 = \{a^n b^n c^k \mid n \geq 0, k \geq 0\}$ и $L_2 = \{a^k b^n c^n \mid n \geq 0, k \geq 0\}$. Очевидно, что L_1 и L_2 — контекстно-свободные языки. Рассмотрим $L_3 = L_1 \cap L_2 = \{a^n b^n c^n \mid n \geq 0\}$. L_3 не является контекстно-свободным по лемме о накачке для контекстно-свободных языков.

Чтобы доказать незамкнутость относительно разности проделаем следующее.

1. Рассмотрим языки $L_4 = \{a^m b^n c^k \mid m \neq n, k \geq 0\}$ и $L_5 = \{a^m b^n c^k \mid n \neq k, m \geq 0\}$. Эти языки являются контекстно-свободными. Это легко заметить, если знать, что язык $L_4' = \{a^m b^n c^k \mid 0 \leq m < n, k \geq 0\}$ задаётся следующей граммтикой:

$$\begin{array}{ll} S \to Sc & T \to aTb \\ S \to T & T \to Tb \\ & T \to b. \end{array}$$

2. Рассмотрим язык $L_6 = \overline{L_6'} = \overline{\{a^n b^m c^k \mid n \geq 0, m \geq 0, k \geq 0\}}$. Данный язык является регулярным.

3. Рассмотрим язык $L_7 = L_4 \cup L_5 \cup L_6$ — контектсно свободный, так как является объединением контекстно-свободных.

4. Рассмотрим $\overline{L_7} = \{a^n b^n c^n \mid n \geq 0\} = L_3$: L_4 и L_5 задают языки с правильным порядком символов, но неравным их количеством, L_6 задаёт язык с неправильным порядком символов. Из пердыдущего пункта мы знаем, что L_3 не является контекстно-свободным.

□

2.7 Конъюнктивные и булевы грамматики

Впервые конъюнктивные и булевы грамматики были предложены Александром Охотиным [53, 54]. Дадим определение конъюнктивной грамматики.

Определение 2.7.1. *Конъюнктивной грамматикой* называется $G = (\Sigma, N, P, S)$, где:

- Σ и N — дизъюнктивные конечные непустые множества терминалов и нетерминалов.
- P — конечное множество продукций, каждая вида

$$A \to \alpha_1 \& ... \& \alpha_n$$

,где $A \in N, n \geq 1$ и $\alpha_1, ..., \alpha_n \in (\Sigma \cup N)^*$.

- $S \in N$ — стартовый нетерминал.

Конъюнктивная грамматика генерирует строки, выводя их из начального символа, так же, как это происходит в контекстно-свободных грамматиках в параграфе 2.1. Промежуточные строки, используемые в процессе вывода, являются формулами следующего вида:

Определение 2.7.2. Пусть $G = (\Sigma, N, P, S)$ — конъюнктивная грамматика. Множество конъюнктивных формул $\mathcal{F}$ определяется индуктивно:

- Пустая строка ε — конъюнктивная формула.
- Любой символ из $(\Sigma \cup N)$ — формула.
- Если $\mathcal{A}$ и $\mathcal{B}$ непустые формулы, тогда $\mathcal{AB}$ — формула.
- Если $\mathcal{A}_1, \ldots, \mathcal{A}_n$ $(n \geqslant 1)$ — формула, тогда $(\mathcal{A}_1 \& \ldots \& \mathcal{A}_n)$ — формула.

Определение 2.7.3. Пусть $G = (\Sigma, N, P, S)$ — конъюнктивная грамматика. Аналогично определению отношения непосредственной выводимости в контекстно-свободной грамматике 2.1.2 определим $\underset{G}{\Rightarrow}$ как отношение непосредственной выводимости на множестве конъюнктивных формул.

- Любой нетерминал в любой формуле может быть перезаписан телом любого правила для этого терминала заключенным в скобки. То есть для любых $s', s'' \in (\Sigma \cup N \cup \{(, \&,)\})^*$ и $A \in N$, таких что $s' A s''$ — формула, и для всех правил вида $A \to \alpha_1 \& \ldots \& \alpha_n \in P$, имеем $s' A s'' \underset{G}{\Rightarrow} s'(\alpha_1 \& \ldots \& \alpha_n) s''$.
- Если формула содержит подформулу в виде конъюнкции одной или нескольких одинаковых терминальных строк, заключенных в скобки, тогда подформула может быть перезаписана терминальной строкой без скобок. То есть для любых $s', s'' \in (\Sigma \cup N \cup \{(, \&,)\})^*$, $(n \geqslant 1)$ и $w \in \Sigma^*$, таких что $s'(w \& \ldots \& w) s''$ — формула, имеем $s'(w \& \ldots \& w) s'' \underset{G}{\Rightarrow} s' w s''$.

Как и в случае контекстно-свободной грамматики обозначим $\underset{G}{\Rightarrow}^*$ рефлексивное транзитивное замыкание отношения $\underset{G}{\Rightarrow}$.

Определение 2.7.4. Пусть $G = (\Sigma, N, P, S)$ — конъюнктивная грамматика. Язык, порождаемый формулой, это множество всех терминальных строк выводимых из этой формулы: $L_G(\mathcal{A}) = \{w \in \Sigma^* \mid \mathcal{A} \underset{G}{\Rightarrow}^* w\}$. Очевидно, что язык порождаемый грамматикой, это язык порождаемый стартовым нетерминалом $S : L(G) = L_G(S) = L(S)$.

Теорема 2.7.1. Пусть $G = (\Sigma, N, P, S)$ — конъюнктивная грамматика. Пусть $\mathcal{A}_1, \ldots, \mathcal{A}_n, \mathcal{B}$ — формулы, $A \in N$, $a \in \Sigma$. Тогда,

1. $L(\varepsilon) = \{\varepsilon\}$.
2. $L(a) = \{a\}$.
3. $L(A) = \bigcup_{A \to \alpha_1 \& \ldots \& \alpha_n \in P} L((\alpha_1 \& \ldots \& \alpha_m))$.
4. $L(\mathcal{A}\mathcal{B}) = L(\mathcal{A}) * L(\mathcal{B})$
5. $L((\mathcal{A}_1 \& \ldots \& \mathcal{A}_n)) = \bigcap_{i=1}^{n} L(\mathcal{A}_i)$.

Теорема 2.7.1 уже подразумевает интерпретацию грамматики как системы уравнений. Используем математический подход, чтобы лучше охарактеризовать конъюнктивные языки с помощью систем уравнений.

Определение 2.7.5 (Выражение)**.** Пусть Σ конечный непустой алфавит. Пусть $X = \{X_1, \ldots, X_N\}$ вектор переменных. Выражение над алфавитом Σ, зависящее от переменных X, определяется индуктивно:

- ε — выражение.
- Любой символ $a \in \Sigma$ — выражение.
- Любая переменная $X_i \in X$ — выражение.
- Если ϕ_1 и ϕ_2 выражения, то $\phi_1\phi_2, (\phi_1 \mid \phi_2), (\phi_1 \& \phi_2)$ также выражения.

Заметим, что любая формула, в терминах определения 2.7.2, является выражением, где нетерминалы формулы это переменные выражения. С другой стороны, любое выражение, не содержащее дизъюнкции, формула.

Предположим, что переменные X_i приняли в качестве значений слова из языка над алфавитом Σ. Определим значение всего выражения.

Определение 2.7.6 (Значение выражения)**.** Пусть $L = (L_1, \ldots, L_n)(L_i \subseteq \Sigma^*)$ вектор из n языков над Σ, где $n \geqslant 1$. Пусть ϕ выражение над Σ, зависящее от переменных $X_1, \ldots, X_n$. Значение выражения ϕ на векторе L — это язык над тем же алфавитом Σ. Обозначим его $\phi(L)$ и определим индуктивно на структуре выражения:

- $\varepsilon(L) = \{\varepsilon\}$.
- $a(L) = \{a\}$ для любого $a \in \Sigma$.
- $X_i(L) = L_i$ для любого $X_i \in X$.
- $\phi_1\phi_2 = \phi_1(L) * \phi_2(L), (\phi_1 \mid \phi_2)(L) = \phi_1(L) \cup \phi_2(L), (\phi_1 \& \phi_2)(L) = \phi_1(L) \cap \phi_2(L)$ для любых выражений ϕ_1 и ϕ_2.

Обобщим определение 2.7.6 на случай вектора выражений.

Определение 2.7.7 (Значение вектора выражений)**.** Пусть $L = (L_1, \ldots, L_n)(L_i \subseteq \Sigma^*)$ вектор из n языков над Σ, где $n \geqslant 1$. Пусть $\phi_1, \ldots, \phi_m$ выражения над Σ, зависящее от переменных $X_1, \ldots, X_n$. Значение вектора выражений $P = (\phi_1, \ldots, \phi_m)$ на векторе L — это вектор языков $P(L) = (\phi_1(L), \ldots, \phi_m(L))$ над тем же алфавитом Σ.

Зададим частичный порядок относительно включения " $\preccurlyeq$ " на множестве языков и расширим его на вектора языков длины n: $(L_1^{'}, \ldots, L_n^{'}) \preccurlyeq (L_1^{''}, \ldots, L_n^{''})$ если и только если $L_i^{'} \subseteq L_i^{''}$ для любого $1 \leqslant i \leqslant n$

Определение 2.7.8. $X = P(X)$ система уравнений над алфавитом Σ и $X = \{X_1, \ldots, X_n\}$, где $P = (\phi_1, \ldots, \phi_n)$ вектор выражений над алфавитом Σ, зависящий от X.

Вектор языков $L = (L_1, \ldots, L_n)$ является решением системы уравнений если $L = P(L)$.

Наименьшее решение L это вектор языков, такой что для любого другого сравнимого вектора языков $L^{'}$ выполняется $L \preccurlyeq L^{'}$.

Заметим, что оператор P на множестве $2^\Sigma \times \ldots \times 2^\Sigma$, что решение системы 2.7.8 это неподвижная точка P и что наименьшее решение системы это наименьшая неподвижная точка оператора P.

Теорема 2.7.2. Для любой системы из определения 2.7.8 с переменными $X_1, \ldots, X_n$, оператор $P = (\phi_1, \ldots, \phi_n)$ имеет наименьшую неподвижную точку $L = (L_1, \ldots, L_n) = \lim_{i \to \infty} P^i \underbrace{(\varnothing, \ldots, \varnothing)}_{n}$.

Приведем пример конъюнктивной грамматики.

Пример 2.7.1 (Пример конъюнктивной грамматики)**.** Следующая конъюнктивная грамматика G порождает язык $\{a^n b^n c^n \mid n \geq 0\}$:

1. $S \to AB\&DC$
2. $A \to aA \mid \varepsilon$
3. $B \to bBc \mid \varepsilon$
4. $C \to cC \mid \varepsilon$
5. $D \to aDb \mid \varepsilon$

Легко видеть, что $L(AB) = \{a^i b^j c^k \mid j = k\}$ и $L(DC) = \{a^i b^j c^k \mid i = j\}$. Тогда $L(S) = L(AB) \cap L(DC) = \{a^n b^n c^n \mid n \geq 0\}$.

В этой грамматике строка abc может быть получена следующим образом. Для начала представим грамматику в виде системы уравнений:

$$\begin{aligned} S &= AB \cap DC \\ A &= \{a\}A \cup \varepsilon \\ B &= \{b\}B\{c\} \cup \varepsilon \\ C &= \{c\}C \cup \varepsilon \\ D &= \{a\}D\{b\} \cup \varepsilon \end{aligned}$$

Используя теорему 2.7.2, будем итеративно вычислять $P^i \underbrace{(\varnothing, \ldots, \varnothing)}_{5}$. На каждом шаге будем подставлять все терминальные строки из языков, порожденных нетерминалами на предыдущем шаге, в соответствующие нетерминалы правой части каждого уравнения и записывать получившиеся терминальные строки в языки нетерминалов текущего шага. Продолжаем до тех пор пока язык, порождаемый нетерминалом S, не будет содержать терминальную строку "abc".

1. На начальном этапе имеем $P^0(\varnothing, \ldots, \varnothing) = (S : \varnothing, A : \varnothing, B : \varnothing, C : \varnothing, D : \varnothing)$

2. Подставляем в первое уравнение терминальные строки из шага 1 в со-

ответствующие нетерминалы, т.е.

$$\begin{aligned} S : \varnothing &= \varnothing\varnothing \cap \varnothing\varnothing \\ A : \{\varepsilon\} &= \{a\}\varnothing \cup \{\varepsilon\} \\ B : \{\varepsilon\} &= \{b\}\varnothing\{c\} \cup \{\varepsilon\} \\ C : \{\varepsilon\} &= \{c\}\varnothing \cup \{\varepsilon\} \\ D : \{\varepsilon\} &= \{a\}\varnothing\{b\} \cup \{\varepsilon\} \end{aligned}$$

В конце итерации получаем $P^1(\varnothing, \ldots, \varnothing) = (S : \varnothing, A : \{\varepsilon\}, B : \{\varepsilon\}, C : \{\varepsilon\}, D : \{\varepsilon\})$

3. Делаем еще одну итерацию,

$$\begin{aligned} S : \{\varepsilon\} &= \{\varepsilon\}\{\varepsilon\} \cap \{\varepsilon\}\{\varepsilon\} \\ A : \{a, \varepsilon\} &= \{a\}\{\varepsilon\} \cup \{\varepsilon\} \\ B : \{bc, \varepsilon\} &= \{b\}\{\varepsilon\}\{c\} \cup \{\varepsilon\} \\ C : \{c, \varepsilon\} &= \{c\}\{\varepsilon\} \cup \{\varepsilon\} \\ D : \{ab, \varepsilon\} &= \{a\}\{\varepsilon\}\{b\} \cup \{\varepsilon\} \end{aligned}$$

В конце итерации получаем $P^2(\varnothing, \ldots, \varnothing) = (S : \{\varepsilon\}, A : \{a, \varepsilon\}, B : \{bc, \varepsilon\}, C : \{c, \varepsilon\}, D : \{ab, \varepsilon\})$

4. Еще одна итерация,

$$\begin{aligned} S : \{\boxed{\text{abc}}, \varepsilon\} &= \{a, \varepsilon\}\{bc, \varepsilon\} \cap \{ab, \varepsilon\}\{c, \varepsilon\} \\ A : \{a, aa, \varepsilon\} &= \{a\}\{a, \varepsilon\} \cup \{\varepsilon\} \\ B : \{bc, bbcc, \varepsilon\} &= \{b\}\{bc, \varepsilon\}\{c\} \cup \{\varepsilon\} \\ C : \{c, cc, \varepsilon\} &= \{c\}\{c, \varepsilon\} \cup \{\varepsilon\} \\ D : \{ab, aabb, \varepsilon\} &= \{a\}\{ab, \varepsilon\}\{b\} \cup \{\varepsilon\} \end{aligned}$$

В конце итерации получили $P^3(\varnothing, \ldots, \varnothing) = (S : \{\boxed{\text{abc}}, \varepsilon\}, A : \{a, aa, \varepsilon\}, B : \{bc, bbcc, \varepsilon\}, C : \{c, cc, \varepsilon\}, D : \{ab, aabb, \varepsilon\})$. Заметим, что терминальная строка "abc" появилась в языке, который порождает стартовый нетерминал S. Т.е. терминальная строка "abc" выводима в грамматике G, что и требовалось показать.

Заметим, что строку "abc" также можно получить применением правил вывода. здесь цифра над стрелкой соответствует номеру примененного пра-

вила.

$$
\begin{aligned}
S &\overset{1}{\Rightarrow} (AB\&DC) \\
&\overset{2}{\Rightarrow} (aAB\&DC) \overset{2}{\Rightarrow} (a\varepsilon B\&DC) \\
&\overset{3}{\Rightarrow} (abBc\&DC) \overset{3}{\Rightarrow} (ab\varepsilon c\&DC) \\
&\overset{5}{\Rightarrow} (abc\&aDbC) \overset{5}{\Rightarrow} (abc\&a\varepsilon bC) \\
&\overset{4}{\Rightarrow} (abc\&abcC) \overset{4}{\Rightarrow} (abc\&abc\varepsilon) \\
&\Rightarrow (abc\&abc) \Rightarrow abc
\end{aligned}
$$

Пример 2.7.2. Конъюнктивная грамматика G для языка $L = \{wcw \mid w \in \{a, b\}^*\}$:

$$
\begin{aligned}
&S \to C\&D \\
&C \to aCa \mid aCb \mid bCa \mid bCb \mid c \\
&D \to aA\&aD \mid bB\&bD \mid cE \\
&A \to aAa \mid aAb \mid bAa \mid bAb \mid cEa \\
&B \to aBa \mid aBb \mid bBa \mid bBb \mid cEb \\
&E \to aE \mid bE \mid \varepsilon
\end{aligned}
$$

Подробнее о конъюнктивных грамматиках можно прочитать в статьях [53, 58, 55, 56].

Дадим определение булевой грамматики.

Определение 2.7.9. *Булевой грамматикой* называется $G = (\Sigma, N, P, S)$, где:

- Σ и N — дизъюнктивные конечные непустые множества терминалов и нетерминалов.
- P — конечное множество продукций, каждая вида

  $$A \to \alpha_1\&...\&\alpha_m\&\neg\beta_1\&...\&\neg\beta_n$$

  ,где $A \in N, m, n >= 0, m + n \geq 1$ и $\alpha_i, \beta_j \in (\Sigma \cup N)^*$.
- $S \in N$ — стартовый нетерминал.

Приведем пример булевой грамматики.

Пример 2.7.3. Следующая булева грамматика порождает язык $\{a^m b^n c^n \mid m, n \geq 0, m \neq n\}$:

$$
\begin{aligned}
&S \to AB\&\neg DC \\
&A \to aA \mid \varepsilon \\
&B \to bBc \mid \varepsilon \\
&C \to cC \mid \varepsilon \\
&D \to aDb \mid \varepsilon
\end{aligned}
$$

Очевидно, что $L(AB) = \{a^m b^n c^n \mid m, n \in \mathbb{N}\}$ и $L(DC) = \{a^n b^n c^m \mid m, n \in \mathbb{N}\}$. Тогда $L(AB) \cap \overline{L(DC)} = \{a^m b^n c^n \mid m, n \geq 0, m \neq n\}$.

Подробнее о булевых грамматиках можно прочитать в статьях [54, 57].

Определим бинарную нормальную форму конъюнктивной грамматики.

Определение 2.7.10 (Бинарная нормальная форма)**.** Конъюнктивная грамматика $G = (\Sigma, N, P, S)$ находится в бинарной нормальной форме, если каждое правило из P имеет вид,

- $A \to B_1 C_1 \& \ldots \& B_m C_m$, где $m \geqslant 1; A, B_i, C_i \in N$.
- $A \to a$, где $A \in N, a \in \Sigma$.
- $S \to \varepsilon$, если только S не содержится в правой части всех правил.

Теорема 2.7.3. Для каждой конъюнктивной грамматики G можно построить конъюнктивную грамматику в бинарной нормальной форме G', такую что $L(G) = L(G')$.

Доказательство теоремы 2.7.3 описано в статье [53].

2.8 Вопросы и задачи

1. Постройте дерево вывода цепочки $w = aababb$ в грамматике $G = \langle\{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b\ S\}, S\rangle$.
2. Постройте все левосторонние выводы цепочки $w = ababab$ в грамматике $G = \langle\{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b \mid S\ S\}, S\rangle$.

3. Постройте все правосторонние выводы цепочки $w = ababab$ в грамматике $G = \langle \{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b \mid S\ S\}, S \rangle$.

4. Постройте все деревья вывода цепочки $w = ababab$ в грамматике $G = \langle \{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b \mid S\ S\}, S \rangle$, соответствующие левосторонним выводам.

5. Постройте все деревья вывода цепочки $w = ababab$ в грамматике $G = \langle \{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b \mid S\ S\}, S \rangle$, соответствующие правосторонним выводам.

Глава 3

Задача о поиске путей с ограничениями в терминах формальных языков

В данной главе сформулируем постановку задачи о поиске путей в графе с ограничениями. А также приведём несколько примеров областей, в которых применяются алгоритмы решения этой задачи.

3.1 Постановка задачи

Пусть нам дан конечный ориентированный помеченный граф $\mathcal{G} = \langle V, E, L \rangle$. Функция $\omega(\pi) = \omega((v_0, l_0, v_1), (v_1, l_1, v_2), \ldots, (v_{n-1}, l_{n-1}, v_n)) = l_0 \cdot l_1 \cdot \ldots \cdot l_{n-1}$ строит слово по пути посредством конкатенации меток рёбер вдоль этого пути. Очевидно, для пустого пути данная функция будет возвращать пустое слово, а для пути длины $n > 0$ — непустое слово длины n.

Если теперь рассматривать задачу поиска путей, то окажется, что то множество путей, которое мы хотим найти, задаёт множество слов, то есть язык. А значит, критерий поиска мы можем сформулировать следующим образом: нас интересуют такие пути, что слова из меток вдоль них принадлежат заданному языку.

Определение 3.1.1. *Задача поиска путей с ограничениями в терминах формальных языков* заключается в поиске множества путей $\Pi = \{\pi \mid \omega(\pi) \in \mathcal{L}\}$.

В задаче поиска путей мы можем накладывать дополнительные ограничения на путь (например, чтобы он был простым, кратчайшим или Эйлеровым [45]), но это уже другая история.

Другим вариантом постановки задачи является задача достижимости.

Определение 3.1.2. *Задача достижимости* заключается в поиске множества пар вершин, для которых найдется путь с началом и концом в этих вершинах, что слово, составленное из меток рёбер пути, будет принадлежать заданному языку. $\Pi' = \{(v_i, v_j) \mid \exists v_i \pi v_j, \omega(\pi) \in \mathcal{L}\}$.

При этом, множество Π может являться бесконечным, тогда как Π' конечно, по причине конечности графа $\mathcal{G}$.

Язык $\mathcal{L}$ может принадлежать разным классам и быть задан разными способами. Например, он может быть регулярным, или контекстно-свободным, или многокомпонентным контекстно-свободным.

Если $\mathcal{L}$ — регулярный, $\mathcal{G}$ можно рассматривать как недетерминированный конечный автомат (НКА), в котором все вершины и стартовые, и конечные. Тогда задача поиска путей, в которой $\mathcal{L}$ — регулярный, сводится к пересечению двух регулярных языков.

Более подробно мы рассмотрим случай, когда $\mathcal{L}$ — контекстно-свободный язык.

Путь $G = \langle \Sigma, N, P \rangle$ — контекстно-свободная граммтика. Будем считать, что $L \subseteq \Sigma$. Мы не фиксируем стартовый нетерминал в определении грамматики, поэтому, чтобы описать язык, задаваемый ей, нам необходимо отдельно зафиксировать стартовый нетерминал. Таким образом, будем говорить, что $L(G, N_i) = \{w | N_i \overset{*}{\underset{G}{\Rightarrow}} w\}$ — это язык задаваемый граммтикой G со стартовым нетерминалом N_i.

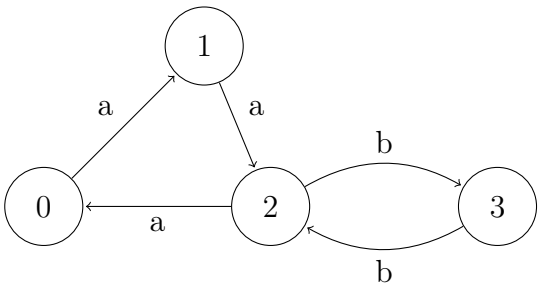
Пример 3.1.1. Пример задачи поиска путей.

Дана грамматика G :

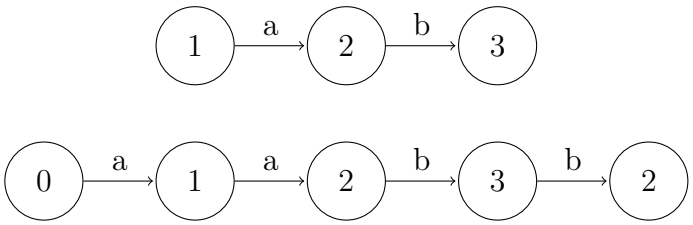
$$S \to ab$$
$$S \to aSb$$

Эта грамматика задаёт язык $\mathcal{L} = a^n b^n$.

И дан граф $\mathcal{G}$:



Тогда примерами путей, принадлежащих множеству $\Pi = \{\pi \mid \omega(\pi) \in \mathcal{L}\}$, являются:



3.2 О разрешимости задачи

Задачи из определения 3.1.1 и 3.1.2 сводятся к построению пересечения языка $\mathcal{L}$ и языка, задаваемого путями графа, R. А мы для обсуждения разрешимости задачи рассмотрим более слабую постановку задачи:

Определение 3.2.1. Необходимо проверить, что существует хотя бы один такой путь π для данного графа, для данного языка $\mathcal{L}$, что $\omega(\pi) \in \mathcal{L}$.

Эта задача сводится к проверке пустоты пересечения языка $\mathcal{L}$ с R — регулярным языком, задаваемым графом. От класса языка $\mathcal{L}$ зависит её разрешимость:

- Если $\mathcal{L}$ регулярный, то получаем задачу пересечения двух регулярных языков:

  $\mathcal{L} \cap R = R'$. R' — также регулярный язык. Проверка регулярного языка на пустоту — разрешимая проблема.

- Если $\mathcal{L}$ контекстно-свободный, то получаем задачу

  $\mathcal{L} \cap R = CF$ — контекстно-свободный. Проверка контекстно-свободного языка на пустоту — разрешимая проблема.

- Помимо иерархии Хомского существуют и другие классификации языков. Так например, класс конъюнктивных (Conj) языков [53] является строгим расширением контекстно-свободных языков и все так же позволяет полиномиальный синтаксический анализ.

  Пусть $\mathcal{L}$ — конъюнктивный. При пересечении конъюнктивного и регулярного языков получается конъюнктивный ($\mathcal{L} \cap R = Conj$), а проблема проверки Conj на пустоту не разрешима [55].

- Ещё один класс языков из альтернативной иерархии, не сравнимой с Иерархией Хомского, — MCFG (multiple context-free grammars) [71]. Как его частный случай — TAG (tree adjoining grammar) [40].

  Если $\mathcal{L}$ принадлежит классу MCFG, то $\mathcal{L} \cap R$ также принадлежит MCFG. Проблема проверки пустоты MCFG разрешима [71].

Существует ещё много других классификаций языков, но поиск универсальной иерархии до сих пор продолжается.

Далее, для изучения алгоритмов решения, нас будет интересовать задача $R \cap CF$.

3.3 Области применения

- Статанализ. Введено Томасом Репсом [63].
- Социальные сети [35].
- RDF обработка [91].
- Биоинформатика [72].
- Применяется для различных межпроцедурных задач [60, 12, 92].
- Графовые БД Впервые предложил Михалис Яннакакис [89].
- OpenCypher [44]
- J.Hellings. CFPQ [34, 36, 35]
- Zhang. CFPQ on rdf graphs [91]
- Bradford [15, 85, 18, 16]

3.4 Вопросы и задачи

1. Пусть есть граф. Задайте грамматику для поиска всех путей, таких, что....
2. Существует ли в графе !!! путь из А в Б, такой что!!!
3. Для графа !!! постройте все пути, удовлетворяющие !!!!

4. Задача 1
5. Задача 2

Глава 4

CYK для вычисления КС запросов

В данной главе мы рассмотрим алгоритм CYK, позволяющий установить принадлежность слова грамматике и предоставить его вывод, если таковой имеется.

Наш главный интерес заключается в возможности применения данного алгоритма для решения описанной в предыдущей главе задачи — поиска путей с ограничениями в терминах формальных языков. Как уже было указано выше, будем рассматривать случай контекстно-свободных языков.

4.1 Алгоритм CYK

Алгоритм CYK (Cocke-Younger-Kasami) — один из классических алгоритмов синтаксического анализа. Его асимптотическая сложность в худшем случае — $O(n^3 * |N|)$, где n — длина входной строки, а N — количество нетерминалов во входной граммтике [39].

Для его применения необходимо, чтобы подаваемая на вход грамматика находилась в Нормальной Форме Хомского (НФХ) 2.4. Других ограничений нет и, следовательно,данный алгоритм применим для работы с произвольными контекстно-своболными языками.

В основе алгоритма лежит принцип динамического программирования. Используются два соображения:

1. Для правила вида $A \to a$ очевидно, что из A выводится ω (с применением этого правила) тогда и только тогда, когда $a = \omega$:

$$A \overset{*}{\Rightarrow} \omega \iff \omega = a$$

2. Для правила вида $A \to BC$ понятно, что из A выводится ω (с применением этого правила) тогда и только тогда, когда существуют две цепочки ω_1 и ω_2 такие, что ω_1 выводима из B, ω_2 выводима из C и при этом $\omega = \omega_1\omega_2$:

$$A \Rightarrow BC \overset{*}{\Rightarrow} \omega \iff \exists \omega_1, \omega_2 : \omega = \omega_1\omega_2, B \overset{*}{\Rightarrow} \omega_1, C \overset{*}{\Rightarrow} \omega_2$$

Или в терминах позиций в строке:

$$A \Rightarrow BC \overset{*}{\Rightarrow} \omega \iff \exists k \in [1 \dots |\omega|] : B \overset{*}{\Rightarrow} \omega[1 \dots k], C \overset{*}{\Rightarrow} \omega[k+1 \dots |\omega|]$$

В процессе работы алгоритма заполняется булева трехмерная матрица M размера $n \times n \times |N|$ таким образом, что

$$M[i, j, A] = true \iff A \overset{*}{\Rightarrow} \omega[i \dots j]$$

.

Первым шагом инициализируем матрицу, заполнив значения $M[i, i, A]$:

- $M[i, i, A] = true$, если в грамматике есть правило $A \to \omega[i]$.
- $M[i, i, A] = false$, иначе.

Далее используем динамику: на шаге $m > 1$ предполагаем, что ячейки матрицы $M[i', j', A]$ заполнены для всех нетерминалов A и пар $i', j' : j' - i' < m$. Тогда можно заполнить ячейки матрицы $M[i, j, A]$, где $j - i = m$ следующим образом:

$$M[i, j, A] = \bigvee_{A \to BC} \bigvee_{k=i}^{j-1} M[i, k, B] \wedge M[k, j, C]$$

По итогу работы алгоритма значение в ячейке $M[0, |\omega|, S]$, где S — стартовый нетерминал грамматики, отвечает на вопрос о выводимости цепочки ω в грамматике.

Пример 4.1.1. Рассмотрим пример работы алгоритма CYK на грамматике правильных скобочных последовательностей в Нормальной Форме Хомского.

$$S \to AS_2 \mid \varepsilon \qquad S_2 \to b \mid BS_1 \mid SS_3 \qquad A \to a$$
$$S_1 \to AS_2 \qquad S_3 \to b \mid BS_1 \qquad B \to b$$

Проверим выводимость цепочки $\omega = aabbab$.

Так как трехмерные матрицы рисовать на двумерной бумаге не очень удобно, мы будем иллюстрировать работу алгоритма двумерными матрицами размера $n \times n$, где в ячейках указано множество нетерминалов, выводящих соответствующую подстроку.

Шаг 1. Инициализируем матрицу элементами на главной диагонали:

$$\begin{pmatrix} \{A\} & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \{A\} & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{A\} & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} \end{pmatrix}$$

Шаг 2. Заполняем диагональ, находящуюся над главной:

$$\begin{pmatrix} \{A\} & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \{A\} & \{S_1\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{A\} & \{S_1\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} \end{pmatrix}$$

В двух ячейках появилисб нетерминалы S_1 благодяря присутствиб в грамматике правила $S_1 \to AS_2$.

Шаг 3. Заполняем следующую диагональ:

$$\begin{pmatrix} \{A\} & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \{A\} & \{S_1\} & \{S_2\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \{S_2, S_3\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{A\} & \{S_1\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} \end{pmatrix}$$

Шаг 4. И следующую за ней:

$$\begin{pmatrix} \{A\} & \varnothing & \varnothing & \{S_1, S\} & \varnothing & \varnothing \\ \varnothing & \{A\} & \{S_1\} & \{S_2\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \{S_2, S_3\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{A\} & \{S_1\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} \end{pmatrix}$$

Шаг 5 Заполняем предпоследнюю диагональ:

$$\begin{pmatrix} \{A\} & \varnothing & \varnothing & \{S_1, S\} & \varnothing & \varnothing \\ \varnothing & \{A\} & \{S_1\} & \{S_2\} & \varnothing & \{S_2\} \\ \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \{S_2, S_3\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{A\} & \{S_1\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} \end{pmatrix}$$

Шаг 6. Завершаем выполнение алгоритма:

$$\begin{pmatrix} \{A\} & \varnothing & \varnothing & \{S_1, S\} & \varnothing & \{S_1, S\} \\ \varnothing & \{A\} & \{S_1\} & \{S_2\} & \varnothing & \{S_2\} \\ \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \{S_2, S_3\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \{A\} & \{S_1\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} \end{pmatrix}$$

Стартовый нетерминал находится в верхней правой ячейке, а значит цепочка $aabbab$ выводима в нашей грамматике.

Пример 4.1.2. Теперь выполним алгоритм на цепочке $\omega = abaa$.

Шаг 1. Инициализируем таблицу:

$$\begin{pmatrix} \{A\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \{A\} & \varnothing \\ \varnothing & \varnothing & \varnothing & \{A\} \end{pmatrix}$$

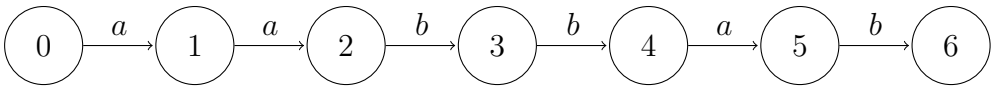
Шаг 2. Заполняем следующую диагональ:

$$\begin{pmatrix} \{A\} & \{S_1, S\} & \varnothing & \varnothing \\ \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \{A\} & \varnothing \\ \varnothing & \varnothing & \varnothing & \{A\} \end{pmatrix}$$

Больше ни одну ячейку в таблице заполнить нельзя и при этом стартовый нетерминал отсутствует в правой верхней ячейке, а значит эта строка не выводится в грамматике правильных скобочных последовательностей.

4.2 Алгоритм для графов на основе CYK

Первым шагом на пути к решению задачи достижимости с использованием CYK является модификация представления входа. Прежде мы сопоставляли каждому символу слова его позицию во входной цепочке, поэтому при инициализации заполняли главную диагональ матрицы. Теперь вместо этого обозначим числами позиции между символами. В результате слово можно представить в виде линейного графа следующим образом(в качестве примера рассмотрим слово *aabbab* из предыдущей главы 4.1):



Что нужно изменить в описании алгоритма, чтобы он продолжал работать при подобной нумерации? Каждая буква теперь идентифицируется не одним числом, а парой — номера слева и справа от нее. При этом чисел стало на одно больше, чем при прежнем способе нумерации.

Возьмем матрицу $(n+1) \times (n+1) \times |N|$ и при инициализации будем заполнять не главную диагональ, а диагональ прямо над ней. Таким образом, мы начинаем наш алгоритм с определения значений $M[i, j, A]$, где $j = i + 1$. При этом наши дальнейшие действия в рамках алгоритма не изменятся.

Для примера 4.1.1 на шаге инициализации матрица выглядит следующим образом:

$$\begin{pmatrix} \varnothing & \{A\} & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \{A\} & \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{A\} & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

А в результате работы алгоритма имеем:

$$\begin{pmatrix} \varnothing & \{A\} & \varnothing & \varnothing & \{S_1, S\} & \varnothing & \{S_1, S\} \\ \varnothing & \varnothing & \{A\} & \{S_1\} & \{S_2\} & \varnothing & \{S_2\} \\ \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} & \varnothing & \{S_2, S_3\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{A\} & \{S_1\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \{B, S_2, S_3\} \\ \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

Мы представили входную строку в виде линейного графа, а на шаге инициализации получили его матрицу смежности. Добавление нового нетерминала в язейку матрицы можно рассматривать как нахождение нового пути между соответствующими вершинами, выводимого из добавленного нетерминала. Таким образом, шаги алгоритма напоминают построение транзитивного замыкания графа. Различие заключается в том, что мы добавляем новые ребра только для тех пар нетерминалов, для которых существует соответстующее правило в грамматике.

Алгоритм можно обобщить и на произвольные графы с метками, рассматриваемые в этом курсе. При этом можно ослабить ограничение на форму входной грамматики: она должна находиться в ослабленной Нормальной Форме Хомского (2.4.3).

Шаг инициализации в алгоритме теперь состоит из двух пунктов.

- Как и раньше, с помощью продукций вида

  $$A \to a, \text{ где } A \in N, a \in \Sigma$$

  заменяем терминалы на ребрах входного графа на множества нетерминалов, из которых они выводятся.

- Добавляем в каждую вершину петлю, помеченную множеством нетерминалов для которых в данной граммтике есть правила вида

  $$A \to \varepsilon, \text{ где } A \in N.$$

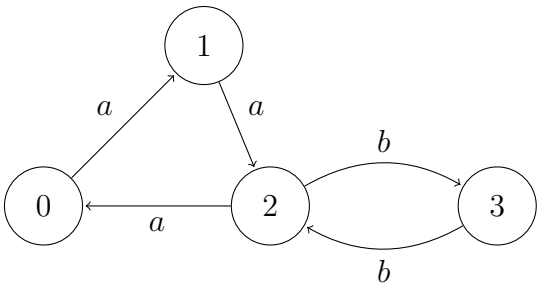
Затем используем матрицу смежности получившегося графа (обозначим ее M) в качестве начального значения. Дальнейший ход алгоритма можно описать псевдокодом, представленным в листинге 2.

Algorithm 2 Алгоритм КС достижимости на основе CYK

```
1: function contextFreePathQuerying(G, 𝒢)
2:     n ← the number of nodes in 𝒢
3:     M ← the modified adjacency matrix of 𝒢
4:     P ← the set of production rules in G
5:     while M is changing do
6:         for k ∈ 0..n do
7:             for i ∈ 0..n do
8:                 for j ∈ 0..n do
9:                     for all N1 ∈ M[i,k], N2 ∈ M[k,j] do
10:                        if N3 → N1N2 ∈ P then
11:                            M[i,j] += {N3}
12:    return M
```

После завершения алгоритма, если в некоторой ячейке результирующей матрицы с номером (i, j) находятся стартовый нетерминал, то это означает, что существует путь из вершины i в вершину j, удовлетворяющий данной грамматике. Таким образом, полученная матрица является ответом для задачи достижимости для заданных графа и граммтики.

Пример 4.2.1. Рассмотрим работу алгоритма на графе

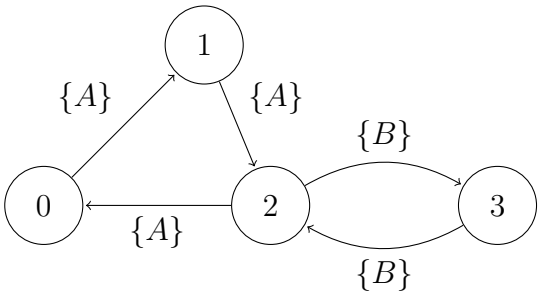


и грамматике:

$$S \to AB \qquad A \to a$$
$$S \to AS_1 \qquad B \to b$$
$$S_1 \to SB$$

Данный пример является классическим и еще не раз будет использоваться в рамках данного курса.

Инициализация. Заменяем терминалы на ребрах графа на нетерминалы, из которых они выводятся, и строим матрицу смежности получившегося графа:



$$\begin{pmatrix} \varnothing & \{A\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \{A\} & \varnothing \\ \{A\} & \varnothing & \varnothing & \{B\} \\ \varnothing & \varnothing & \{B\} & \varnothing \end{pmatrix}$$

Итерация 1. Итерируемся по k, i и j, пытаясь найти пары нетерминалов, для которых существуют правила вывода, их выводящие. Нам интересны следующие случаи:

- $k = 2, i = 1, j = 3 : A \in M[1,2], B \in M[2,3]$, так как в грамматике присутствует правило $S \to AB$, добавляем нетерминал S в ячейку $M[1,3]$.
- $k = 3, i = 1, j = 2 : S \in M[1,3], B \in M[3,2]$, поскольку в грамматике есть правило $S_1 \to SB$, добавляем нетерминал S_1 в ячейку $M[1,2]$.

В остальных случаях либо какая-то из клеток пуста, либо не существует продукции в грамматике, выводящей данные два нетерминала.

Матрица после данной итерации:

$$\begin{pmatrix} \varnothing & \{A\} & \varnothing & \varnothing \\ \varnothing & \varnothing & \{A, \boldsymbol{S_1}\} & \{S\} \\ \{A\} & \varnothing & \varnothing & \{B\} \\ \varnothing & \varnothing & \{B\} & \varnothing \end{pmatrix}$$

Итерация 2. Снова итерируемся по k, i, j. Рассмотрим случаи:

- $k = 1, i = 0, j = 2 : A \in M[0,1], S_1 \in M[1,2]$, так как в грамматике присутствует правило $S \to AS_1$, добавляем нетерминал S в ячейку $M[0,2]$.

- $k = 2, i = 0, j = 3 : S \in M[0,2], B \in M[2,3]$, поскольку в грамматике есть правило $S_1 \to SB$, добавляем нетерминал S_1 в ячейку $M[0,3]$.

Матрица на данном шаге:

$$\begin{pmatrix} \varnothing & \{A\} & \{S\} & \{S_1\} \\ \varnothing & \varnothing & \{A, S_1\} & \{S\} \\ \{A\} & \varnothing & \varnothing & \{B\} \\ \varnothing & \varnothing & \{B\} & \varnothing \end{pmatrix}$$

Итерация 3. Рассматриваемые на данном шаге случаи:

- $k = 0, i = 2, j = 3 : A \in M[2,0], S_1 \in M[0,3]$, так как в грамматике присутствует правило $S \to AS_1$, добавляем нетерминал S в ячейку $M[2,3]$.

- $k = 3, i = 2, j = 2 : S \in M[2,3], B \in M[3,2]$, поскольку в грамматике есть правило $S_1 \to SB$, добавляем нетерминал S_1 в ячейку $M[2,2]$.

Матрица после этой итерации:

$$\begin{pmatrix} \varnothing & \{A\} & \{S\} & \{S_1\} \\ \varnothing & \varnothing & \{A, S_1\} & \{S\} \\ \{A\} & \varnothing & \{S_1\} & \{B, \boldsymbol{S}\} \\ \varnothing & \varnothing & \{B\} & \varnothing \end{pmatrix}$$

Итерация 4. Рассмариваемые случаи:

- $k = 2, i = 1, j = 2 : A \in M[1,2], S_1 \in M[2,2]$, так как в грамматике присутствует правило $S \to AS_1$, добавляем нетерминал S в ячейку $M[1,2]$.

- $k = 2, i = 1, j = 3 : S \in M[1,2], B \in M[2,3]$, поскольку в грамматике есть правило $S_1 \to SB$, добавляем нетерминал S_1 в ячейку $M[1,3]$.

Матрица:

$$\begin{pmatrix} \varnothing & \{A\} & \{S\} & \{S_1\} \\ \varnothing & \varnothing & \{A, \boldsymbol{S}, S_1\} & \{S, \boldsymbol{S_1}\} \\ \{A\} & \varnothing & \{S_1\} & \{B, S\} \\ \varnothing & \varnothing & \{B\} & \varnothing \end{pmatrix}$$

Итерация 5. Рассмотрим на это шаге:

- $k = 1, i = 0, j = 3 : A \in M[0,1], S_1 \in M[1,3]$, поскольку в грамматике есть правило $S \to AS_1$, добавляем нетерминал S в ячейку $M[0,3]$.

- $k = 3, i = 0, j = 2 : S \in M[0,3], B \in M[3,2]$, поскольку в грамматике есть правило $S_1 \to SB$, добавляем нетерминал S_1 в ячейку $M[0,2]$.

Матрица на этой итерации:

$$\begin{pmatrix} \varnothing & \{A\} & \{S, \boldsymbol{S_1}\} & \{\boldsymbol{S}, S_1\} \\ \varnothing & \varnothing & \{A, S, S_1\} & \{S, S_1\} \\ \{A\} & \varnothing & \{S_1\} & \{B, S\} \\ \varnothing & \varnothing & \{B\} & \varnothing \end{pmatrix}$$

Итерация 6. Интересующие нас на этом шаге случаи:

- $k = 0, i = 2, j = 2 : A \in M[2,0], S_1 \in M[0,2]$, поскольку в грамматике есть правило $S \to AS_1$, добавляем нетерминал S в ячейку $M[2,2]$.

- $k = 2, i = 2, j = 3 : S \in M[2,2], B \in M[2,3]$, поскольку в грамматике есть правило $S_1 \to SB$, добавляем нетерминал S_1 в ячейку $M[2,3]$.

Матрица после данного шага:

$$\begin{pmatrix} \varnothing & \{A\} & \{S, S_1\} & \{S, S_1\} \\ \varnothing & \varnothing & \{A, S, S_1\} & \{S, S_1\} \\ \{A\} & \varnothing & \{\boldsymbol{S}, S_1\} & \{B, S, \boldsymbol{S_1}\} \\ \varnothing & \varnothing & \{B\} & \varnothing \end{pmatrix}$$

На следующей итерации матрица не изменяется, поэтому заканчиваем работу алгоритма. В результате, если ячейка $M[i, j]$ содержит стартовый нетерминал S, то существует путь из i в j, удовлетворяющий ограничениям, заданным грамматикой.

Можно заметить, что мы делаем много лишних итераций. Можно переписать алгоритм так, чтобы он не просматривал заведомо пустые ячейки. Данную модификацию предложил Й.Хеллингс в работе [34], также она реализована в работе [91].

Псевдокод алгоритма Хеллингса представлен в листинге 3.

Algorithm 3 Алгоритм Хеллингса

```
1: function contextFreePathQuerying(G = ⟨Σ, N, P, S⟩, 𝒢 = ⟨V, E, L⟩)
2:     r ← {(N_i, v, v) | v ∈ V ∧ N_i → ε ∈ P} ∪ {(N_i, v, u) | (v, t, u) ∈ E ∧ N_i → t ∈ P}
3:     m ← r
4:     while m ≠ ∅ do
5:         (N_i, v, u) ← m.pick()
6:         for (N_j, v', v) ∈ r do
7:             for N_k → N_j N_i ∈ P таких что ((N_k, v', u) ∉ r) do
8:                 m ← m ∪ {(N_k, v', u)}
9:                 r ← r ∪ {(N_k, v', u)}
10:        for (N_j, u, v') ∈ r do
11:            for N_k → N_i N_j ∈ P таких что ((N_k, v, v') ∉ r) do
12:                m ← m ∪ {(N_k, v, v')}
13:                r ← r ∪ {(N_k, v, v')}
14:    return r
```

Пример 4.2.2. Запустим алгоритм Хеллингса на нашем примере.

Инициализация

$$m = r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2)\}$$

Итерации внешнего цикла. Будем считеть, что r и m — упорядоченные списки и *pick* возвращает его голову, оставляя хвост. Новые элементы добавляются в конец.

1. Обрабатываем $(A, 0, 1)$. Ни один из вложенных циклов не найдёт новых путей, так как для рассматриваемого ребра есть только две возможности достроить путь: $2 \xrightarrow{A} 0 \xrightarrow{A} 1$ и $0 \xrightarrow{A} 1 \xrightarrow{A} 2$ и ни одна из соответствующих строк не выводтся в заданной граммтике.

2. Перед началом итерации

$$m = \{(A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2)\},$$

r не изменилось. Обрабатываем $(A, 1, 2)$. В данной ситуации второй цикл найдёт тройку $(B, 2, 3)$ и соответсвующее правило $S \to A\ B$. Это значит, что и в m и в r добавится тройка $(S, 1, 3)$.

3. Перед началом итерации

$$m = \{(A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3)\}.$$

Обрабатываем $(A, 2, 0)$. Внутринние циклы ничего не найдут, новых путей н появится.

4. Перед началом итерации

$$m = \{(B, 2, 3), (B, 3, 2), (S, 1, 3)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3)\}.$$

Обрабатываем $(B, 2, 3)$. Первый цикл мог бы найти $(A, 1, 2)$, однако при проверке во вложенном цикле выяснится, что $(S, 1, 3)$ уже найдена. В итоге, на данной итерации новых путей н появится.

5. Перед началом итерации

$$m = \{(B, 3, 2), (S, 1, 3)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3)\}.$$

Обрабатываем $(B, 3, 2)$. Первый цикл найдёт $(S, 1, 3)$ и соответствующее правило $S_1 \to S\ B$. Это значит, что и в m и в r добавится тройка $(S_1, 1, 2)$.

6. Перед началом итерации

$$m = \{(S, 1, 3), (S_1, 1, 2)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2)\}.$$

Обрабатываем $(S, 1, 3)$. Второй цикл мог бы найти $(B, 3, 2)$, однако при проверке во вложенном цикле выяснится, что $(S_1, 1, 2)$ уже найдена. В итоге, на данной итерации новых путей н появится.

7. Перед началом итерации

$$m = \{(S_1, 1, 2)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2)\}.$$

Обрабатываем $(S_1, 1, 2)$. Первый цикл найдёт $(A, 0, 1)$ и соответствующее правило $S \to A\ S_1$. Это значит, что и в m и в r добавится тройка $(S, 0, 2)$.

8. Перед началом итерации

$$m = \{(S, 0, 2)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2)\}.$$

Обрабатываем $(S, 0, 2)$. Найдено: $(B, 2, 3)$ и соответствующее правило $S_1 \to S\ B$. В m и в r добавится тройка $(S_1, 0, 3)$.

9. Перед началом итерации

$$m = \{(S_1, 0, 3)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2), (S_1, 0, 3)\}.$$

Обрабатываем $(S_1, 0, 3)$. Найдено: $(A, 2, 0)$ и соответствующее правило $S \to A\ S_1$. В m и в r добавится тройка $(S, 2, 3)$.

10. Перед началом итерации

$$m = \{(S, 2, 3)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2), (S_1, 0, 3), (S, 2, 3)\}.$$

Обрабатываем $(S, 2, 3)$. Найдено: $(B, 3, 2)$ и соответствующее правило $S_1 \to S\ B$. В m и в r добавится тройка $(S_1, 2, 2)$.

11. Перед началом итерации

$$m = \{(S_1, 2, 2)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2), (S_1, 0, 3), (S, 2, 3), (S_1, 2, 2)\}.$$

Обрабатываем $(S_1, 2, 2)$. Найдено: $(A, 1, 2)$ и соответствующее правило $S \to A\ S_1$. В m и в r добавится тройка $(S, 1, 2)$.

12. Перед началом итерации

$$m = \{(S, 1, 2)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2), (S_1, 0, 3), (S, 2, 3), (S_1, 2, 2), (S, 1, 2)\}.$$

Обрабатываем $(S, 1, 2)$. Найдено: $(B, 2, 3)$ и соответствующее правило $S_1 \to S\ B$. В m и в r добавится тройка $(S_1, 1, 3)$.

13. Перед началом итерации

$$m = \{(S_1, 1, 3)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2), (S_1, 0, 3), (S, 2, 3), (S_1, 2, 2), (S, 1, 2), (S_1, 1, 3)\}.$$

Обрабатываем $(S_1, 1, 3)$. Найдено: $(A, 0, 1)$ и соответствующее правило $S \to A\ S_1$. В m и в r добавится тройка $(S, 0, 3)$.

14. Перед началом итерации

$$m = \{(S, 0, 3)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2), (S_1, 0, 3), (S, 2, 3), (S_1, 2, 2), (S, 1, 2), (S_1, 1, 3), (S, 0, 3)\}.$$

Обрабатываем $(S, 0, 3)$. Найдено: $(B, 3, 2)$ и соответствующее правило $S_1 \to S\ B$. В m и в r добавится тройка $(S_1, 0, 2)$.

15. Перед началом итерации

$$m = \{(S_1, 0, 2)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2), (S_1, 0, 3), (S, 2, 3), (S_1, 2, 2), (S, 1, 2), (S_1, 1, 3), (S, 0, 3), (S_1, 0, 2)\}.$$

Обрабатываем $(S_1, 0, 2)$. Найдено: $(A, 2, 0)$ и соответствующее правило $S \to A\ S_1$. В m и в r добавится тройка $(S, 2, 2)$.

16. Перед началом итерации

$$m = \{(S, 2, 2)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2), (S_1, 0, 3), (S, 2, 3), (S_1, 2, 2), (S, 1, 2), (S_1, 1, 3), (S, 0, 3), (S_1, 0, 2), (S, 2, 2)\}.$$

Обрабатываем $(S, 2, 2)$. Найдено: $(B, 2, 3)$ и соответствующее правило $S_1 \to S\ B$. В m и в r добавится тройка $(S_1, 2, 3)$.

17. Перед началом итерации

$$m = \{(S_1, 2, 3)\},$$

$$r = \{(A, 0, 1), (A, 1, 2), (A, 2, 0), (B, 2, 3), (B, 3, 2), (S, 1, 3), (S_1, 1, 2), (S, 0, 2), (S_1, 0, 3), (S, 2, 3), (S_1, 2, 2), (S, 1, 2), (S_1, 1, 3), (S, 0, 3), (S_1, 0, 2), (S, 2, 2), (S_1, 2, 3)\}.$$

Обрабатываем $(S_1, 2, 3)$. Могло бы быть найдено: $(A, 1, 2)$ и соответствующее правило $S \to A\ S_1$, однако тройка $(S, 1, 3)$ уже есть в r. А значит никаких новых троек найдено не будет и m становится пустым. Это была последняя итерация внешнего цикла, в r на текущий момент уже содержится всё ршение.

Как можно заметить, количество итераций внешнего цикла также получилось достаточно большим. Проверьте, зависит ли оно от порядка обработки элементов из m. При этом внутренние циклы в нашем случае достаточно короткие, так как просматриваются только “существенные” элементы и избегается дублирование.

4.3 Вопросы и задачи

1. Проверить работу алгоритма CYK для цепочек на грамматике

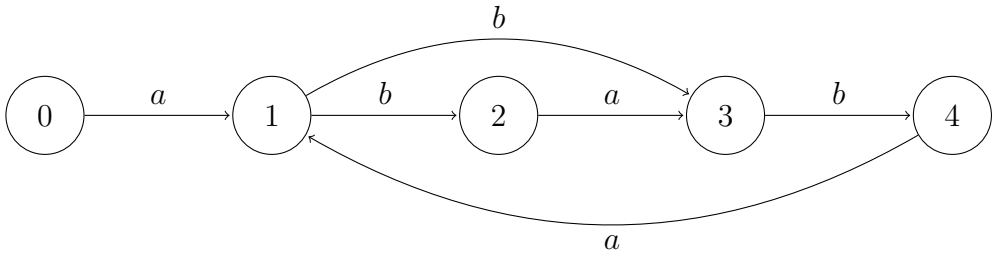
$E \to E + E$
$E \to E * E$
$E \to (E)$
$E \to n$

и словах (алфавит $\Sigma = \{n, +, *, (,)\}$)

$(n + n) * n$
$n + n * n$
$n + n + n + n$
$n + (n * n) + n$

2. Изучить вычислительную сложность алгоритма CYK для матриц в зависимости от размера входного графа (размер грамматики считать фиксированным).

3. Проверить работу алгоритма CYK для графов на графе



И грамматике

$S \to SS$
$S \to AB$
$A \to a$
$B \to b$

4. Оцените временную сложность алгоритма Хеллингса и сравните её с оценкой для наивного обобщения CYK.

Глава 5

КС и конъюнктивная достижимость через произведение матриц

В данном разделе мы рассмотрим алгоритм решения задачи контекстно-свободной и конъюнктивной достижимости, основанный на произведении матриц. Будет показано, что при использовании конъюнктивных граммтик, представленный алгоритм находит переапроксимацию истинного решения задачи.

5.1 КС достижимость через произведение матриц

В главе 4.2 был изложен алгоритм для решения задачи КС достижимости на основе CYK. Заметим, что обход матрицы напоминает умножение матриц в ячейках которых множества нетерминалов:

$$M_3 = M_1 \times M_2$$
$$M_3[i,j] = \sum_{k=1}^{n} M[i,k] * M[k,j]$$

, где сумма — это объединение множеств:

$$\sum_{k=1}^{n} = \bigcup_{k=1}^{n}$$

, а поэлементное умножение определено следующим образом:

$$S_1 * S_2 = \{N_1^0 ... N_1^m\} * \{N_2^0 ... N_2^l\} = \{N_3 \mid (N_3 \to N_1^i N_2^j) \in P\}.$$

Таким образом, алгоритм решения задачи КС достижимости может быть выражена в терминах перемножения матриц над полукольцом с соответствующими операциями.

Для частного случая этой задачи, синтаксического анализа линейного входа, существует алгоритм Валианта [81], использующий эту идею. Однако он не обобщается на графы из-за того, что существенно использует возможность упорядочить обход матрицы (см. разницу в CYK для линейного случая и для графа). Поэтому, хотя для линейного случая алгоритм Валианта является алгоритмом синтаксического анализа для произвольных КС граммтик за субкубическое время, его обобщение до задачи КС достижимости в произвольных графах с сохранением асимптотики является нетривиальной задачей [89]. В настоящее время алгоритм с субкубической сложностью получен только для частного случая — языка Дика с одним типом скобок — Филипом Брэдфорlом [17].

В случае с линейным входом, отдельного внимания заслуживает работа Лиллиан Ли (Lillian Lee) [47], где она показывает, что задача перемножения матриц сводима к синтаксическому анализу линейного входа. Аналогичных результатов для графов на текущий момент не известно.

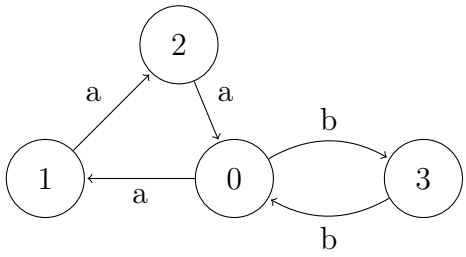
Поэтому рассмотрим более простую идею, изложенную в статье Рустама Азимова [7]: будем строить транзитивное замыкание графа через наивное (не через возведение в квадрат) умножение матриц.

Пусть $\mathcal{G} = (V, E)$ — входной граф и $G = (N, \Sigma, P)$ — входная грамматика. Тогда алгоритм может быть сформулирован как представлено в листинге 4.

Algorithm 4 Context-free recognizer for graphs

```
1:  function contextFreePathQuerying(𝒢, G)
2:      n ← количество узлов в 𝒢
3:      E ← направленные ребра в 𝒢
4:      P ← набор продукций из G
5:      T ← матрица n × n, в которой каждый элемент ∅
6:      for all (i, x, j) ∈ E do                    ▷ Инициализация матрицы
7:          T_{i,j} ← T_{i,j} ∪ {A | (A → x) ∈ P}
8:      while матрица T меняется do
9:          T ← T ∪ (T × T)                  ▷ Вычисление транзитивного замыкания
10:     return T
```

Пример 5.1.1 (Пример работы)**.** Пусть есть граф $\mathcal{G}$:



и грамматика G:

$$\begin{aligned} S &\to AB & A &\to a \\ S &\to AS_1 & B &\to b \\ S_1 &\to SB \end{aligned}$$

Пусть T_i — матрица, полученная из T после применения цикла, описанного в строках **8-9** алгоритма 4, i раз. Тогда T_0, полученная напрямую из графа, выглядит следующим образом:

$$T_0 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \varnothing & \varnothing & \{A\} & \varnothing \\ \{A\} & \varnothing & \varnothing & \varnothing \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

Далее показано получение матрицы T_1.

$$T_0 \times T_0 = \begin{pmatrix} \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \varnothing \\ \varnothing & \varnothing & \varnothing & \{S\} \\ \varnothing & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_1 = T_0 \cup (T_0 \times T_0) = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \varnothing & \varnothing & \{A\} & \varnothing \\ \{A\} & \varnothing & \varnothing & \{\boldsymbol{S}\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

После первой итерации цикла нетерминал в ячейку $T[2,3]$ добавился нетерминал S. Это означает, что существует такой путь π из вершины 2 в вершину 3 в графе $\mathcal{G}$, что $S \xrightarrow{*} \omega(\pi)$. В данном примере путь состоит из двух ребер $2 \xrightarrow{a} 0$ и $0 \xrightarrow{b} 3$, так что $S \xrightarrow{*} ab$.

Вычисление транзитивного замыкания заканчивается через k итераций, когда достигается неподвижная точка процесса: $T_{k-1} = T_k$. Для данного примера $k = 13$, так как $T_{13} = T_{12}$. Весь процесс работы алгоритма (все матрицы T_i) показан ниже (на каждой итерации новые элементы выделены жирным).

$$T_2 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \varnothing & \varnothing & \{A\} & \varnothing \\ \{A, \boldsymbol{S_1}\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_3 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \{\boldsymbol{S}\} & \varnothing & \{A\} & \varnothing \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_4 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \{S\} & \varnothing & \{A\} & \{\boldsymbol{S_1}\} \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_5 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B, \boldsymbol{S}\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_6 = \begin{pmatrix} \{\boldsymbol{S_1}\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_7 = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1, \boldsymbol{S}\} & \varnothing & \varnothing & \{S\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_8 = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, \boldsymbol{S_1}\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_9 = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S\} & \varnothing & \{A\} & \{S_1, \boldsymbol{S}\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_{10} = \begin{pmatrix} \{S_1\} & \{A\} & \varnothing & \{B, S\} \\ \{S, \boldsymbol{S_1}\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_{11} = \begin{pmatrix} \{S_1, \boldsymbol{S}\} & \{A\} & \varnothing & \{B, S\} \\ \{S, S_1\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

$$T_{12} = \begin{pmatrix} \{S_1, S\} & \{A\} & \varnothing & \{B, S, \boldsymbol{S_1}\} \\ \{S, S_1\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix} \qquad T_{13} = \begin{pmatrix} \{S_1, S\} & \{A\} & \varnothing & \{B, S, S_1\} \\ \{S, S_1\} & \varnothing & \{A\} & \{S_1, S\} \\ \{A, S_1, S\} & \varnothing & \varnothing & \{S, S_1\} \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

Таким образом, результат алгоритма 4 для нашего примера — это матрица $T_{13} = T_{12}$. Заметим, что для данного алгоритма приведённый пример также является худшим случаем: на каждой итерации в матрицу добавляется ровно один нетерминал, при том, что необходимо заполнить порядка $O(n^2)$ ячеек.

5.1.1 Расширение алгоритма для конъюнктивных грамматик

Матричный алгоритм для конъюнктивных грамматик отличается от алгоритма 4 для контекстно-свободных грамматик только операцией умножения матриц, в остальном алгоритм остается без изменений. Определим операцию умножения матриц.

Определение 5.1.1. Пусть M_1 и M_2 матрицы размера n. Определим операцию $\circ$ сдедующим образом:

$$M_1 \circ M_2 = M_3,$$

$$M_3[i,j] = \{A \mid \exists (A \to B_1C_1 \& \ldots \& B_mC_m) \in P, (B_k, C_k) \in d[i,j] \forall k = 1, \ldots, m\}$$

, где

$$d[i,j] = \bigcup_{k=1}^{n} M_1[i,k] \times M_2[k,j].$$

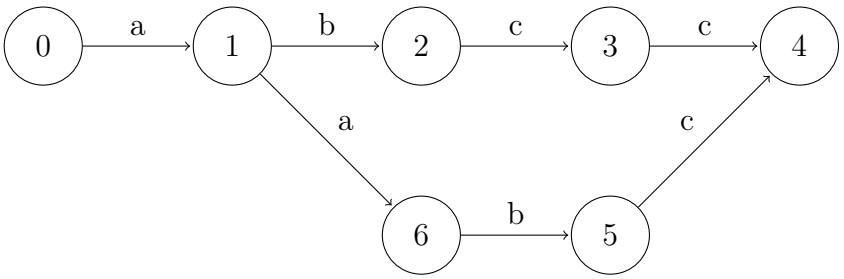
Важно заметить, что алгоритм для конъюнктивных грамматик, в отличие от алгоритма для контекстно-свободных грамматик, дает лишь верхнюю оценку ответа. То есть все нетерминалы, которые должны быть в ячейках матрицы результата, содержатся там, но вместе с ними содержатся и лишние нетерминалы. Рассмотрим пример, иллюстрирующий появление лишних нетерминалов.

Пример 5.1.2. Грамматика G:

$$S \to AB\&DC \qquad C \to c$$
$$A \to a \qquad D \to DC \mid b$$
$$B \to BC \mid b$$

Очевидно, что грамматика G задает язык из одного слова $L(G) = \{abc\} = \{abc^*\} \cap \{a^*bc\}$.

Пусть есть граф $\mathcal{G}$:



Применяя алгоритм, получим, что существует путь из вершины 0 в вершину 4, выводимый из нетерминала S. Однако очевидно, что в графе такого пути нет. Такое поведение алгоритма наблюдается из-за того, что существует путь “abcc”, соответствующий $L(AB) = \{abc^*\}$ и путь “aabc”, соответствующий $L(DC) = \{a^*bc\}$, но они различны. Однако алгоритм не может это проверить,

так как оперирует понятием достижимости между вершинами, а не наличием различных путей. Более того, в общем случае для конъюнктивных граммтик такую проверку реализовать нельзя. Поэтому для классической семантики достидимости с ограничениями в терминах конъюнктивных граммтик результат работы алгоритма является оценкой сверху.

Существует альтернативная семантика, когда мы трактуем конъюнкцию в праой части правил как крнъюнкцию в Datalog (подробнее о Datalog в параграфе 12). Т.е если есть правило $S \to AB\&DC$, то должен быть путь принадлежащий языку $L(AB)$ и путь принадлежащий языку $L(DC)$. В такой семантике алгоритм дает точный ответ.

Подробнее алгоритм описан в статье Рустама Азимова и Семёна Григорьева [8]. Стоит также отметить, что обобщения данного алогритма для булевых грамматик не существует, хотя и сущетсвует частное решение для случая, когда граф не содержит циклов (является DAG-ом), предложенное Екатериной Шеметовой [73].

5.2 Особенности реализации

Алгоритмы, описанные выше, удобны с точки зрения реализации тем, что могут быть эффективно реализованы с использованием высокопроизводительных библиотек линейной алгебры, которые эксплуатируют возможности параллельных вычислений на современных CPU и GPGPU [51]. Это позволяет с минимальными затратими получить эффективную параллельную реализацию алгоритма для решения задачи КС достижимости в графах. Благодаря этому, хотя асимптотически приведенные алгоритмы имеют большую сложность чем, скажем, алгоритм Хеллингса, в результате эффективного распараллеливания на практике они работают быстрее однопоточных алгоритов с лучшей сложностью.

Далее рассмотрим некоторые детали, упрощающие реализацию с использованием современных библиотек и аппаратного обеспечения.

Так как множество нетерминалов и правил конечно, то мы можем свести представленный выше алгоритм к булевым матрицам: для каждого нетерминала заведём матрицу, такую что в ячейке стоит 1 тогда и только тогда, когда в исходной матрице в соответствующей ячейке сожержится этот нетерминал. Тогда перемножение пары таких матриц, соответсвующих нетерминалам A и B, соответствует построению путей, выводимых из нетерминалов, для которых есть правила с правой частью вида AB.

Пример 5.2.1. Представим в виде набора булевых матриц следующую матрицу:

$$T_0 = \begin{pmatrix} \varnothing & \{A\} & \varnothing & \{B\} \\ \varnothing & \varnothing & \{A\} & \varnothing \\ \{A\} & \varnothing & \varnothing & \varnothing \\ \{B\} & \varnothing & \varnothing & \varnothing \end{pmatrix}$$

Тогда:

$$T_{0\_A} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad T_{0\_B} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Тогда при наличии правила $S \to AB$ в грамматике получим:

$$T_{1\_S} = T_{0\_A} \times T_{0\_B} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Алгоритм же может быть переформулирован так, как показано в листинге 6. Такой взгляд на алгоритм позволяет исполььдвать для его реализации существующие высокорпоизводительные библиотеки для работы с булевыми матрицами (например M4RI[1] [3]) или библиотеки для линейной алгебры (например CUSP [24]).

Listing 5 Context-free path quering algorithm. Boolean matrix version

1: **function** EVALCFPQ($D = (V, E), G = (N, \Sigma, P)$)
2: $n \leftarrow |\mathrm{V}|$
3: $T \leftarrow \{T^{A_i} \mid A_i \in N, T^{A_i}$ is a matrix $n \times n$, $T^{A_i}_{k,l} \leftarrow$ `false`$\}$
4: **for all** $(i, x, j) \in E$, $A_k \mid A_k \to x \in P$ **do** $T^{A_k}_{i,j} \leftarrow$ `true`
5: **for** $A_k \mid A_k \to \varepsilon \in P$ **do** $T^{A_k}_{i,i} \leftarrow$ `true`
6: **while** any matrix in T is changing **do**
7: **for** $A_i \to A_j A_k \in P$ **do** $T^{A_i} \leftarrow T^{A_i} + (T^{A_j} \times T^{A_k})$
8: **return** T

[1]M4RI — одна из высокопроизводительных библиотек для работы с логическими матрицами на CPU. Реализует Метод Четырёх Русских. Исходный код библиотеки: `https://bitbucket.org/malb/m4ri/src/master/`. Дата посещения: 30.03.2020.

С другой стороны, для запросов, выразимых в терминах граммтик с небольшим количеством нетерминалов, практически может быть выгодно представлять множества нетерминалов в ячейке матрицы в виде битового вектора следующим образом. Нумеруем все нетерминалы с нуля, в векторе стоит 1 на позиции i, если в множестве есть нетерминал с номером i. Таким образом, в каждой ячейке хранится битовый вектор длины $|N|$. Тогда операция умножения определяется следующим образом:

$$v_1 \times v_2 = \{v \mid \exists (v, v_3) \in P, \mathit{append}(v_1, v_2) \& v_3 = v_3\},$$

где & — побитовое **“и”**.

Правила надо кодировать соответственно: продукция это пара, где первый элемент — битовый вектор длины $|N|$ с единственной единицей в позиции, соответствующей нетерминалу в правой части, а второй элемент — вектор длины $2|N|$, с двумя единицами кодирующими первый и второй нетерминалы, соответственно.

Пример 5.2.2. Пусть $N = \{S, A, B\}$ и в грамматике есть продукция $S \to AB$. Тогда занумеруем нетерминалы $(S, 0), (A, 1), (B, 2)$. Продукция примет вид $[1, 0, 0] \to [0, 1, 0, 0, 0, 1]$. Матрица T_0 примет вид (здесь “.” означает, что в ячейке стоит $[0, 0, 0]$):

$$T_0 = \begin{pmatrix} . & [0,1,0] & . & [0,0,1] \\ . & . & [0,1,0] & . \\ [0,1,0] & . & . & . \\ [0,0,1] & . & . & . \end{pmatrix}$$

После выполнения умножения получим:

$$T_1 = T_0 + T_0 \times T_0 = \begin{pmatrix} . & [0,1,0] & . & [0,0,1] \\ . & . & [0,1,0] & . \\ [0,1,0] & . & . & [1,0,0] \\ [0,0,1] & . & . & . \end{pmatrix}$$

На практике в роли векторов могут выступать беззнаковые целые числа. Например, 32 бита под ячейки в матрице и 64 бита под правила (или 8 и 16, если позволяет количество нетерминалов в грамматике, или 16 и 32). Тогда умножение выражается через битовые операции и сравнение, что довольно эффективно с точки зрения вычислений.

Для небольших запросов такой подход к реализации может оказаться быстрее — в данном случае скорость зависит от деталей. Минус подхода в том,

что нет возможности использовать готовые библиотеку ленейной алгебры без предварительной модификации. Только если они не являются параметризуемыми и не позволяют задать собственный тип и собственную операцию умножения и сложения (иными словами, собственное полукольцо). Такую возможность предусматривает, например, стандарт GraphBLAS[2] и, соответственно, его реализации, такие как SuiteSparce[3] [25].

Также стоит замеить, что при работе с реальными графами матрицы как правило оказываются разреженными, а значит необходимо использовать соответствующие представления матриц (CRS, покоординатное, Quad Tree [28]) и библиотеки, работающие с таким представлениями.

Однако даже при использовании разреженных матриц, могут возникнуть проблемы с размером реальных данных и объёмом памяти. Напрмиер, для вычислений на GPGPU лучше всего, когда все нужные для вычисления матрицы помещаются на одну карту. Тогда можно свести обмен данными сежду хостом и графическим сопроцессором к мимнимуму. Если не помещаются все, то нужно, чтобы помещалясь хотя бы тройка непостредственно обрабатываемых матриц (два операнда и результат). В самом тяжёлом случае в памяти не удаётся раместить даже операнды целиком и тогда приходится прибегать к поблочному умнодению матриц.

Отдельной инженерной проблемой является масштабирование алгоритмов на несколько вычислительных узлов, как на несколько CPU, так и на несколько GPGPU.

Важным свойством рассмотренного алгоритма является то, что описанные проблемы с объёмом памяти и масштабированием могут эффективно решаться в рамках библиотек линейной алгебры общего назначения, что избавляет от необходимости создавать специализированные решения для конкетных задач.

5.3 Вопросы и задачи

1. Находить кратчайшие пути в графах, используя идеи из секции 5.1.

2. Превратить граф, использующийся для CFPQ, в дерево.

[2]GraphBLAS — открытый стандарт, описывающий набор примитивов и операций, необходимый для реализации графовых алгоритмов в терминах лнейной алгебры. Web-страница проекта: `https://github.com/gunrock/graphblast`. Дата доступа: 30.03.2020.

[3]SuteSparse — это специализированное программное обеспечения для работы с разреженными матрицами, которое включает в себя реализацию GraphBLAS API. Web-страница проекта: `http://faculty.cse.tamu.edu/davis/suitesparse.html`. Дата доступа: 30.03.2020.

3. Реализовать предложенные идеи на различных архитектурах.

4. Замерить производительность и расходы памяти по сравнению с существующими реализациями.

Глава 6

КС достижимость через тензорное произведение

Предыдущий подход позволяет выразить задачу поиска путей с ограничениями в терминах формальных языков через набор матричных операций. Это позволяет использовать высокопроизводительные библиотеки, массовопараллельные архитектуры и другие готовые решения для линейной алгебры. Однако, такой подход требует, чтобы грамматика находилась в ослабленной нормальной форме Хомского, что приводит к её разрастанию. Можно ли как-то избежать этого?

В данном разделе мы предложим альтернативное сведение задачи поиска путей к матричным операциям. В результате мы сможем избежать преобразования грамматики в ОНФХ, однако, матрицы, с которыми нам предётся работать, будут существенно бóльшего размера.

В основе подхода лежит использование рекурсивных сетей или рекурсивных автоматов в качестве представления контекстно-свободных грамматик и использование тензорного (прямого) произведения для нахождения пересечения автоматов.

6.1 Рекурсивные автоматы и сети

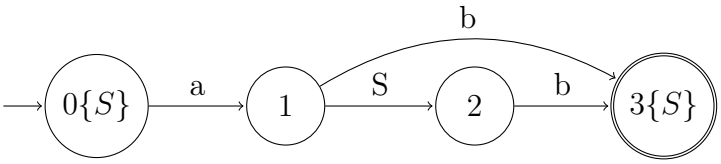
Рекурсивный автомат или сеть — это представление контекстно-свободных грамматик, обобщающее конечные автоматы. В нашей работе мы будем придерживаться термина **рекурсивный автомат**. Классическое определение рекурсивного автомата выглядит следующим образом.

Определение 6.1.1. Рекурсивный автомат — это кортеж вида $\langle N, \Sigma, S, D \rangle$, где

- N — нетерминальный алфавит;
- Σ — терминальный алфавит;
- S — стартовый нетерминал;
- D — конечный автомат над $N \cup \Sigma$ в котором стартовые и финальные состояния помечены подмножествами N.

Построим рекурсивный автомат для грамматики G:

$$S \to aSb$$
$$S \to ab$$



(6.1)

Используем стандартные обозначения для стартовых и финальных состояний. Дополнительно в стартовых и финальных состояниях укажем нетерминалы, для которых эти состояния стартовые/финальные.

В некоторых случаях рекурсивный автомат можно рассматривать как конечный автомат над смешанным алфавитом. Именно такой взгляд мы будем использовать при изложении алгоритма.

6.2 Тензорное произведение

Тензорное произведение матриц или произведение Кронекера — это бинарная операция, обозначаемая $\otimes$ и определяемая следующим образом.

Определение 6.2.1. Пусть даны две матрицы: A размера $m \times n$ и B размера $p \times q$. Произведение Кронекера или тензорное произведение матриц A и B — это блочная матрица C размера $mp \times nq$, вычисляемая следующим образом:

$$C = A \otimes B = \begin{pmatrix} A_{0,0}B & \cdots & A_{0,n-1}B \\ \vdots & \ddots & \vdots \\ A_{m-1,0}B & \cdots & A_{m-1,n-1}B \end{pmatrix}$$

Пример 6.2.1.

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \otimes \begin{pmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} = \begin{pmatrix} 1 \begin{pmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} & 2 \begin{pmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \\ 3 \begin{pmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} & 4 \begin{pmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \end{pmatrix} =$$

$$= \left(\begin{array}{cccc|cccc} 5 & 6 & 7 & 8 & 10 & 12 & 14 & 16 \\ 9 & 10 & 11 & 12 & 18 & 20 & 22 & 24 \\ 13 & 14 & 15 & 16 & 26 & 28 & 30 & 32 \\ \hline 15 & 18 & 21 & 24 & 20 & 24 & 28 & 32 \\ 27 & 30 & 33 & 36 & 36 & 40 & 44 & 48 \\ 39 & 42 & 45 & 48 & 52 & 56 & 60 & 64 \end{array}\right) \qquad (6.2)$$

Заметим, что для определения тензорного произведения матриц достаточно определить операцию умножения на элементах исходных матриц. Также отметим, что произведение Кронекера не является коммутативным. При этом всегда существуют две матрицы перестоновок P и Q такие, что $A \otimes B = P(B \otimes A)Q$. Это свойство потребуется нам в дальнейшем.

Теперь перейдём к графам. Сперва дадим классическое определение тензорного произведения двух неориентированных графов.

Определение 6.2.2. Пусть даны два графа: $\mathcal{G}_1 = \langle V_1, E_1 \rangle$ и $\mathcal{G}_2 = \langle V_2, E_2 \rangle$. Тензорным произведением этих графов будем называть граф $\mathcal{G}_3 = \langle V_3, E_3 \rangle$, где $V_3 = V_1 \times V_2$, $E_3 = \{((v_1, v_2), (u_1, u_2)) \mid (v_1, u_1) \in E_1 \text{ и } (v_2, u_2) \in E_2\}$.

Иными словами, тензорным произведением двух графов является граф, такой что:

1. множество вершин — это прямое произведение множеств вершин исходных графов;

2. ребро между вершинами $v = (v_1, v_2)$ и $u = (u_1, u_2)$ существует тогда и только тогда, когда существуют рёбра между парами вершин v_1, u_1 и v_2, u_2 в соответсвующих графах.

Для того, чтобы построить тензорное произведение ориентированных графов, необходимо в предыдущем определении, в условии существования ребра в результирующем графе, дополнительно потребовать, чтобы направления рёбер совпадали. Данное требование получается естесвенным образом, если считать, что пары вершин, задающие ребро, упорядочены, поэтому формальное определение отличаться не будет.

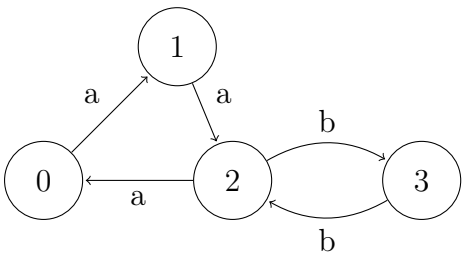
Осталось добавить метки к рёбрам. Это приведёт к логичному усилению требованя к существованию ребра: метки рёбер в исходных графах должны совпадать. Таким образом, мы получаем следующее определение тензорного произведения ориентированных графов с метками на рёбрах.

Определение 6.2.3. Пусть даны два ориентированных графа с метками на рёбрах: $\mathcal{G}_1 = \langle V_1, E_1, L_1 \rangle$ и $\mathcal{G}_2 = \langle V_2, E_2, L_2 \rangle$. Тензорным произведением этих графов будем называть граф $\mathcal{G}_3 = \langle V_3, E_3, L_3 \rangle$, где $V_3 = V_1 \times V_2$, $E_3 = \{((v_1, v_2), l, (u_1, u_2)) \mid (v_1, l, u_1) \in E_1 \text{ и } (v_2, l, u_2) \in E_2\}$, $L_3 = L_1 \cap L_2$.

Нетрудно заметить, что матрица смежности графа $\mathcal{G}_3$ равна тензорному произведению матриц смежностей исходных графов $\mathcal{G}_1$ и $\mathcal{G}_2$.

Пример 6.2.2. Рассмотрим пример. В качестве одного из графов возьмём рекурсивный автомат, построенный ранее (изображение 6.1). Его матрица смежности выглядит следующим образом.

$$M_1 = \begin{pmatrix} . & [a] & . & . \\ . & . & [S] & [b] \\ . & . & . & [b] \\ . & . & . & . \end{pmatrix}$$



(6.3)

Второй граф представлен на изображении 6.3. Его матрица смежности имеет следующий вид.

$$M_2 = \begin{pmatrix} . & [a] & . & . \\ . & . & [a] & . \\ [a] & . & . & [b] \\ . & . & [b] & . \end{pmatrix}$$

Теперь вычислим $M_1 \otimes M_2$.

$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . \\ . & . & [S] & [b] \\ . & . & . & [b] \\ . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & [a] & . & . \\ . & . & [a] & . \\ [a] & . & . & [b] \\ . & . & [b] & . \end{pmatrix} =$$

$$= \left(\begin{array}{cccc|cccc|cccc} . & . & . & . & . & [a] & . & . & . & . & . & . \\ . & . & . & . & . & . & [a] & . & . & . & . & . \\ . & . & . & . & [a] & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & [b] \\ . & . & . & . & . & . & . & . & . & . & [b] & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & [b] \\ . & . & . & . & . & . & . & . & . & . & [b] & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \end{array}\right)$$

6.3 Алгоритм

Идея алгоритма основана на обобщении пересечения двух конечных автоматов до пересечения рекурсивного автомата, построенного по грамматике, со входным графом.

Пересечение двух конечных автоматов — тензорное произведение соответствующих графов. Пересечение языков коммутативно, тензорное произведение нет, но, как было сказано в разделе 6.2, существует решение этой проблемы.

Будем рассматривать два конечных автомата: одни получан из входного графа, второй из грамматики. Можно найти их пересечение, вычислив тензорное произведение матриц смежности соответствующих графов. Однако, одной

такой итерации не достаточно для решения исходной задачи. За первую итерацию мы найдём только те пути, которые выводятся в граммтике за одни шаг. После этого необходимо добавить соответствующие рёбра во входной граф и повторить операцию: так мы найдём пути, выводимые за два шага. Данные действия надо повторять до тех пор, пока не перестанут находиться новые пары достижимых вершин. Псевдокод, описывающий данные действия, представлен в листинге 6.

Listing 6 Поиск путей через тензорное произведение

```
1: function contextFreePathQueryingTP(G, 𝒢)
2:     R ← рекурсивный автомат для G
3:     N ← нетерминальный алфавит для R
4:     S ← стартовые состояния для R
5:     F ← конечные состояния для R
6:     M_1 ← матрица смежности R
7:     M_2 ← матрица смежности 𝒢
8:     for N_i ∈ N do
9:         if N_i →* ε then
10:            for all j ∈ 𝒢.V : M_2[j, j] ← M_2[j, j] ∪ {N_i}   ▷ Добавим петли для нетерминалов, выводящих ε
11:    while матрица M_2 изменяется do
12:        M_3 ← M_1 ⊗ M_2                                      ▷ Пересечение графов
13:        tC_3 ← transitiveClosure(M_3)
14:        n ← количество строк и столбцов матрицы M_3   ▷ размер матрицы M_3 = n × n
15:        for i ∈ 0..n do
16:            for j ∈ 0..n do
17:                if tC_3[i, j] then
18:                    s ← стартовая вершина ребра tC_3[i, j]
19:                    f ← конечная вершина ребра tC_3[i, j]
20:                    if s ∈ S and f ∈ F then
21:                        x, y ← getCoordinates(i, j)
22:                        M_2[x, y] ← M_2[x, y] ∪ {getNonterminals(s, f)}
23:    return M_2
```

Алгоритм исполняется до тех пор, пока матрица смежности M_2 изменяется. На каждой итерации цикла алгоритм последовательно проделывает следующие команды: пересечение двух автоматов через тензорное произведение, транзитивное замыкание результата тензорного произведения и итерация по всем ячейкам получившейся после транзитивного замыкания матрицы, что

необходимо для поиска новых пар достижимых вершин. Во время итерации по ячейкам матрицы транзитивного замыкания алгоритм сначала проверяет наличие ребра в данной ячейке, а затем — принадлежность стартовой и конечной вершин ребра к стартовому и конечному состоянию входного рекурсивного автомата. При удовлетворении этих условий алгоритм добавляет нетерминал (или нетерминалы), соответствующие стартовой и конечной вершинам ребра, в ячейку матрицы M_2, полученной с благодаря функции $getCoordinates(i, j)$.

Представленный алгоритм не требует преобразования граммтики в ОНФХ, более того, рекурсивный автомат может быть минимизирован. Однако, результатом тензорного рпоизведения является матрица существенно бóльшего размера, чем в алгоритме, основанном на матричном рпоизведении. кроме этого, необходимо искать транзитивное замыкание этой матрицы.

Ещё одним важным свойством представленного алгоритма является его оптимальность при обработке регулярных запросов. Так как по контекстно свободной граммтике мы не можем пределить, задаёт ли она регулярный язык, то при добавлении в язык запросов возмодности задавать контекстно-свободные ограничения, возникает проблема: мы не можем в общем случае отличить регулярный запрос от контекстно-свободного. Следовательно, мы вынуждены применять наиболее общий механизм выполнения заросов, что может приводить к существенным накладным расходам при выполнении регулярного запроса. Данный же алгоритм не выполнит лишних действий, так как сразу выполнит классическое пересечение двух автоматов и получит результат.

6.4 Примеры

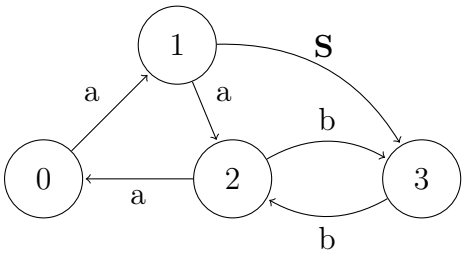
Рассмотрим подробно ряд примеров работы описанного алгоритма. Будем для каждой итерации внешнего цикла выписывать результаты основных операций: тензорного произведения, транзитивного замыкания, обновления матрицы смежности входного графа. Новые, по сравнению с предыдущим состоянием, элементы матриц будем выделять.

Пример 6.4.1. Теоретически худший случай. Такой же как и для матричного.

Итерация 1 (конец). Начало в разделе 6.2, где мы вычислили тензорное произведение матриц смежности. Теперь нам осталось только вычислить транзитивное замыкание полученной матрицы:

$$tc(M_3) = \left(\begin{array}{c|c|c|c} \cdots & \cdot\ [a]\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \cdots & \cdot\ \cdot\ [a]\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \mathbf{[ab]} \\ \cdots & [a]\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \hline \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ [b] \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ [b]\ \cdot \\ \hline \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ [b] \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ [b]\ \cdot \\ \hline \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \\ \cdots & \cdot\ \cdot\ \cdot\ \cdot & \cdots & \cdot\ \cdot\ \cdot\ \cdot \end{array}\right).$$

Мы видим, что в результате транзитивного замыкания появилось новое ребро с меткой ab из вершины $(0, 1)$ в вершину $(3, 3)$. Так как вершина 0 является стартовой в рекурсивном автомате, а 3 является финальной, то слово вдоль пути из вершины 1 в вершину 3 во входном графе выводимо из нетерминала S. Это означает, что в графе должно быть добавлено ребро из 0 в 3 с меткой S, после чего граф будет выглядеть следующим образом:



Матрица смежности обновлённого графа:

$$M_2 = \begin{pmatrix} \cdot & [a] & \cdot & \cdot \\ \cdot & \cdot & [a] & \mathbf{[S]} \\ [a] & \cdot & \cdot & [b] \\ \cdot & \cdot & [b] & \cdot \end{pmatrix}$$

Итерация закончена. Возвращаемся к началу цикла и вновь вычисляем тензорное произведение.

Итерация 2. Вычисляем тензорное произведение матриц смежности.

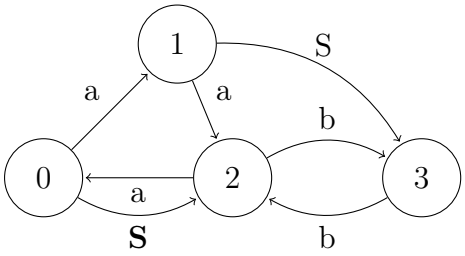
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} \cdot & [a] & \cdot & \cdot \\ \cdot & \cdot & [S] & [b] \\ \cdot & \cdot & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} \cdot & [a] & \cdot & \cdot \\ \cdot & \cdot & [a] & [S] \\ [a] & \cdot & \cdot & [b] \\ \cdot & \cdot & [b] & \cdot \end{pmatrix} =$$

$$= \left(\begin{array}{cccc|cccc|cccc} \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \mathbf{[S]} & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}\right)$$

Вычисляем транзитивное замыкание полученной матрицы:

$$tc(M_3) = \left(\begin{array}{cccc|cccc|cccc} \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \mathbf{[aS]} & \cdot & \cdot & \mathbf{[aSb]} & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [ab] \\ \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \cdot & \cdot & \mathbf{[Sb]} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}\right)$$

В транзитивном замыкании появилось три новых ребра, однако только одно из них соединяет вершины, соответствующие стартовому и конечному состоянию входного рекурсивного автомата. Таким образом только это ребро должно быть добавлено во входной граф. В итоге, обновлённый граф:



И его матрица смежности:

$$M_2 = \begin{pmatrix} . & [a] & [S] & . \\ . & . & [a] & [S] \\ [a] & . & . & [b] \\ . & . & [b] & . \end{pmatrix}$$

Итерация 3. Снова начинаем с тензорного произведения.

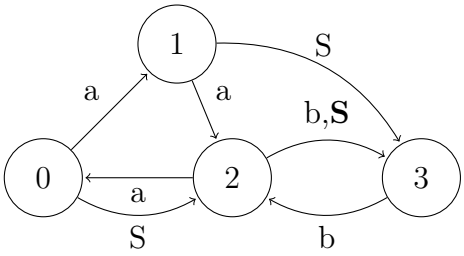
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . \\ . & . & [S] & [b] \\ . & . & . & [b] \\ . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & [a] & [S] & . \\ . & . & [a] & [S] \\ [a] & . & . & [b] \\ . & . & [b] & . \end{pmatrix} =$$

$$= \left(\begin{array}{cccc|cccc|cccc} . & . & . & . & . & [a] & . & . & . & . & . & . \\ . & . & . & . & . & . & [a] & . & . & . & . & . \\ . & . & . & . & [a] & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & [S] & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & [b] \\ . & . & . & . & . & . & . & . & . & . & [b] & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & [b] \\ . & . & . & . & . & . & . & . & . & . & [b] & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \end{array}\right)$$

Затем вычисляем транзитивное замыкание:

$$tc(M_3) = \left(\begin{array}{cccc|cccc|cccc}
. & . & . & . & . & [a] & . & . & . & . & [aS] & . & . & [aSb] & . & .\\
. & . & . & . & . & . & [a] & . & . & . & . & . & . & . & [ab] & .\\
. & . & . & . & [a] & . & . & . & . & \mathbf{[aS]} & . & . & . & . & \mathbf{[aSb]} & .\\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .\\
\hline
. & . & . & . & . & . & . & . & . & [S] & . & . & . & . & \mathbf{[Sb]} & .\\
. & . & . & . & . & . & . & . & . & . & [S] & . & . & [Sb] & . & .\\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & [b] & .\\
. & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . & .\\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .\\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .\\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & [b] & .\\
. & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . & .\\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .\\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .\\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .\\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right)$$

И наконец обновляем граф:



Матрица смежности обновлённого графа:

$$M_2 = \begin{pmatrix} . & [a] & [S] & . \\ . & . & [a] & [S] \\ [a] & . & . & [b, \mathbf{S}] \\ . & . & [b] & . \end{pmatrix}$$

Уже можно заметить закономерность: на каждой итерации мы добавляем ровно одно новое ребро во входной граф, ровно как и в алгоритме, основанном на матричном произведении, и как в алгоритме Хеллингса. То есть находим ровно одну новую пару вешин, между которыми существует интересующий нас путь. Попробуйте спрогонозировать, сколько итераций нам ещё осталось сделать.

Итерауия 4. Продолжаем. Вычисляем тензорное произведение.

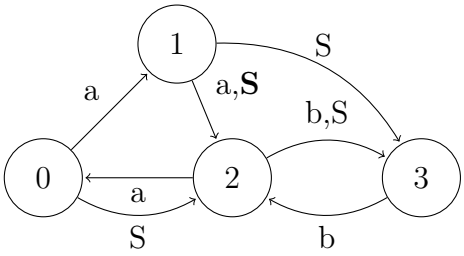
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . \\ . & . & [S] & [b] \\ . & . & . & [b] \\ . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & [a] & [S] & . \\ . & . & [a] & [S] \\ [a] & . & . & [b, S] \\ . & . & [b] & . \end{pmatrix} =$$

$$= \left(\begin{array}{cccc|cccc|cccc} . & . & . & . & . & [a] & . & . & . & . & . & . \\ . & . & . & . & . & . & [a] & . & . & . & . & . \\ . & . & . & . & [a] & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & [b] \\ . & . & . & . & . & . & . & . & . & [b] & . & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & [b] \\ . & . & . & . & . & . & . & . & . & [b] & . & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \end{array}\right)$$

Затем транзитивное замыкание:

$$tc(M_3) = \left(\begin{array}{cccc|cccc|cccc} . & . & . & . & . & [a] & . & . & . & . & [aSb] & . \\ . & . & . & . & . & . & [a] & . & . & . & \mathbf{[aSb]} & [ab] \\ . & . & . & . & [a] & . & . & . & [aS] & . & . & [aSb] \\ . & . & . & . & . & . & . & . & . & . & . & . \\ \hline . & . & . & . & . & . & . & . & [S] & . & . & [Sb] \\ . & . & . & . & . & . & . & . & . & [S] & [Sb] & . \\ . & . & . & . & . & . & . & . & . & [S] & \mathbf{[Sb]} & [b] \\ . & . & . & . & . & . & . & . & . & . & [b] & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & [b] \\ . & . & . & . & . & . & . & . & . & . & [b] & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \end{array}\right)$$

И снова обновляем граф, так как в транзитивном замыкании появился один (и снова ровно один) новый элемент, соответствующий принимающему пути в автомате.



Матрица смежности обновлённого графа:

$$M_2 = \begin{pmatrix} . & [a] & [S] & . \\ . & . & [a, \mathbf{S}] & [S] \\ [a] & . & . & [b, S] \\ . & . & [b] & . \end{pmatrix}$$

Итерация 5. Приступаем к выполнению следующей итерации основного цикла. Вычисляем тензорное произведение.

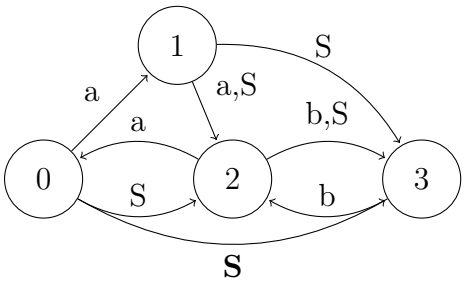
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . \\ . & . & [S] & [b] \\ . & . & . & [b] \\ . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & [a] & [S] & . \\ . & . & [a, S] & [S] \\ [a] & . & . & [b, S] \\ . & . & [b] & . \end{pmatrix} =$$

$$= \left(\begin{array}{cccc|cccc|cccc} . & . & . & . & . & [a] & . & . & . & . & . & . \\ . & . & . & . & . & . & [a] & . & . & . & . & . \\ . & . & . & . & [a] & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ \hline . & . & . & . & . & . & . & [S] & . & . & . & . \\ . & . & . & . & . & . & . & \mathbf{[S]} & [S] & . & . & . \\ . & . & . & . & . & . & . & . & [S] & . & . & [b] \\ . & . & . & . & . & . & . & . & . & . & [b] & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & [b] \\ . & . & . & . & . & . & . & . & . & . & [b] & . \\ \hline . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . \end{array}\right)$$

Затем вычисляем транзитивное замыкание:

$$tc(M_3) = \left(\begin{array}{cccc|cccc|cccc}
. & . & . & . & . & [a] & . & . & . & \mathbf{[aS]} & [aS] & . & . & [aSb] & \mathbf{[aSb]} \\
. & . & . & . & . & . & [a] & . & . & . & [aS] & . & . & [aSb] & [ab] \\
. & . & . & . & [a] & . & . & . & . & [aS] & . & . & . & . & [aSb] \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & [S] & . & . & . & . & [Sb] \\
. & . & . & . & . & . & . & . & . & [S] & [S] & . & . & [Sb] & \mathbf{[Sb]} \\
. & . & . & . & . & . & . & . & . & . & [S] & . & . & [Sb] & [b] \\
. & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & [b] \\
. & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right)$$

Обновляем граф:



Матрица смежности обновлённого графа:

$$M_2 = \begin{pmatrix} . & [a] & [S] & \mathbf{[S]} \\ . & . & [a,S] & [S] \\ [a] & . & . & [b,S] \\ . & . & [b] & . \end{pmatrix}$$

Итерация 6. И наконец последняя содержательная итерация основного цикла.

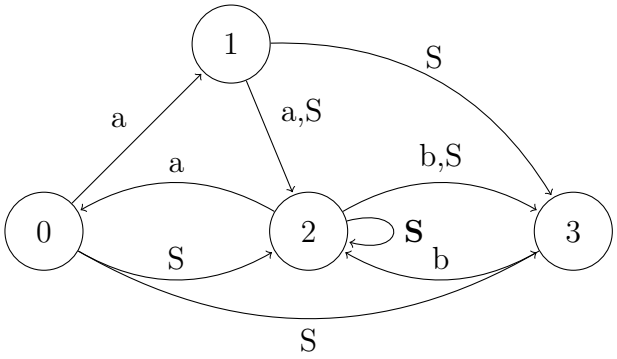
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} \cdot & [a] & \cdot & \cdot \\ \cdot & \cdot & [S] & [b] \\ \cdot & \cdot & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} \cdot & [a] & [S] & [S] \\ \cdot & \cdot & [a,S] & [S] \\ [a] & \cdot & \cdot & [b,S] \\ \cdot & \cdot & [b] & \cdot \end{pmatrix} =$$

$$= \left(\begin{array}{cccc|cccc|cccc} \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \mathbf{[S]} & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & [S] & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}\right)$$

Транзитивное замыкание:

$$tc(M_3) = \left(\begin{array}{cccc|cccc|cccc} \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & [aS] & [aS] & \cdot & \cdot & [aSb] & [aSb] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & [aS] & \cdot & \cdot & [aSb] & [ab] \\ \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & [aS] & \mathbf{[aS]} & \cdot & \cdot & \mathbf{[aSb]} & [aSb] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \mathbf{[S]} & \cdot & \cdot & \mathbf{[Sb]} & [Sb] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & [S] & \cdot & \cdot & [Sb] & [Sb] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \cdot & \cdot & [Sb] & [b] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [b] & \cdot \\ \hline \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}\right)$$

Обновлённый граф:



И матрица смежности:

$$M_2 = \begin{pmatrix} . & [a] & [S] & [S] \\ . & . & [a,S] & [S] \\ [a] & . & \mathbf{[S]} & [b,S] \\ . & . & [b] & . \end{pmatrix}$$

Следующая итерация не приведёт к изменению графа. Читатель может убедиться в этом самостоятельно. Соответственно, алгоритм можно завершать. Нам потребовалось семь итераций (шесть содержательных и одна проверочная), на каждой из которых вычисляются тензорное произведение двух матриц смежности и транзитивное замыкание результата.

Матрица смежности получилась такая же, как и раньше, ответ правильный. Мы видим, что количество итераций внешнего цикла такое же как и у алгоритма CYK (пример 4.2.1).

Пример 6.4.2. В данном примере мы увидим, как структура грамматики и, соответственно, рекурсивного автомата, влияет на процесс вычислений.

Интуитивно понятно, что чем меньше состояний в рекурсивной сети, тем лучше. То есть желательно получить как можно более компактное описание контекстно-свободного языка.

Для примера возьмём в качестве КС языка язык Дика на одном типе скобок и опишем его двумя различными грамматиками. Первая граммтика классическая:

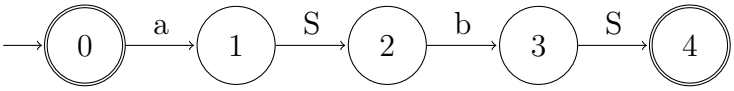
$$G_1 = \langle \{a,\ b\}, \{S\}, \{S \to a\ S\ b\ S \mid \varepsilon\} \rangle$$

Во второй грамматике мы будем использовать конструкции регулярных выражений в правой части правил. То есть вторая грамматика находитмся в EBNF (Расширенная форма Бэкуса-Наура [37, 87]).

$$G_2 = \langle \{a,\ b\}, \{S\}, \{S \to (a\ S\ b)^*\} \rangle$$

Построим рекурсивные автоматы N_1 и N_2 и их матрицы смежности для этих грамматик.

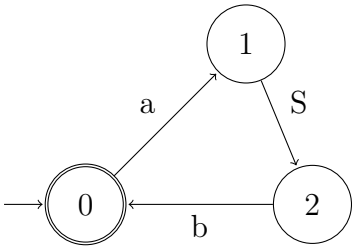
Рекурсивный автомат N_1 для грамматики G_1:



Матрица смежности N_1:

$$M_1^1 = \begin{pmatrix} . & [a] & . & . & . \\ . & . & [S] & . & . \\ . & . & . & [b] & . \\ . & . & . & . & [S] \\ . & . & . & . & . \end{pmatrix}$$

Рекурсивный автомат N_2 для грамматики G_2:

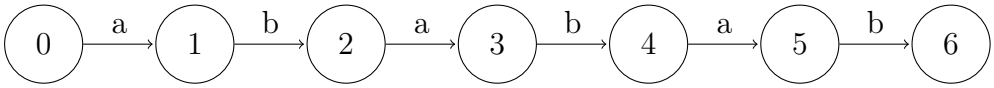


Матрица смежности N_2:

$$M_1^2 = \begin{pmatrix} . & [a] & . \\ . & . & [S] \\ [b] & . & . \end{pmatrix}$$

Первое очевидное наблюдение — количество состояний в N_2 меньше, чем в N_1. Это значит, что матрица смежности, а значит, и результат тензорного произведения будут меньше, следовательно, вычисления будут производиться быстрее.

Выполним пошагово алгоритм для двух заданных рекурсивных сетей на линейном входе:



Сразу дополним матрицу смежности нетерминалами, выводящими пустую строку, и получим следующую матрицу:

$$M_2 = \begin{pmatrix} [S] & [a] & . & . & . & . & . & . \\ . & [S] & [b] & . & . & . & . & . \\ . & . & [S] & [a] & . & . & . & . \\ . & . & . & [S] & [b] & . & . & . \\ . & . & . & . & [S] & [a] & . & . \\ . & . & . & . & . & [S] & [b] & . \\ . & . & . & . & . & . & [S] & [b] \\ . & . & . & . & . & . & . & [S] \end{pmatrix}$$

Сперва запустим алгоритм на данном входе и N_2. Первый шаг первой итерации — вычисление тензорного произведения $M_1^2 \otimes M_2$.

$$M_3 = M_1^2 \otimes M_2 = \begin{pmatrix} . & [a] & . \\ . & . & [S] \\ [b] & . & . \end{pmatrix} \otimes \begin{pmatrix} [S] & [a] & . & . & . & . & . & . \\ . & [S] & [b] & . & . & . & . & . \\ . & . & [S] & [a] & . & . & . & . \\ . & . & . & [S] & [b] & . & . & . \\ . & . & . & . & [S] & [a] & . & . \\ . & . & . & . & . & [S] & [b] & . \\ . & . & . & . & . & . & [S] & [b] \\ . & . & . & . & . & . & . & [S] \end{pmatrix} =$$

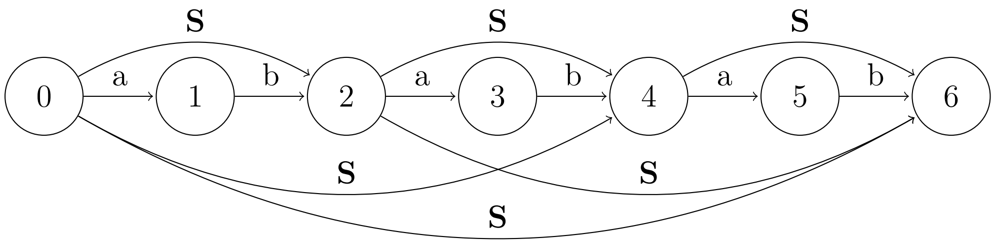
$$= \left(\begin{array}{cccccccc|cccccccc} . & . & . & . & . & . & . & . & . & [a] & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & [a] & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & [a] & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ \hline . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] \\ \hline . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & [b] & . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & [b] & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & [b] & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \end{array}\right)$$

Опустим промежуточные шаги вычисления транзитивного замыкания M_3 и сразу представим конечный результат:

$$tc(M_3) = \left(\begin{array}{ccccccc|ccccccc|ccccccc}
\cdot & \cdot & \mathbf{[aSb]} & \cdot & \mathbf{[aSbaSb]} & \cdot & \mathbf{[aSbaSbaSb]} & \cdot & [a] & \cdot & \mathbf{[aSba]} & \cdot & \mathbf{[aSbaSba]} & \cdot & \cdot & \mathbf{[aS]} & \cdot & \mathbf{[aSbaS]} & \cdot & \mathbf{[aSbaSbaS]} & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \mathbf{[aSb]} & \cdot & \mathbf{[aSbaSb]} & \cdot & \cdot & \cdot & [a] & \cdot & \mathbf{[aSba]} & \cdot & \cdot & \cdot & \cdot & \mathbf{[aS]} & \cdot & \mathbf{[aSbaS]} & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \mathbf{[aSb]} & \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \mathbf{[aS]} & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] \\
\hline
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & [b] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & [b] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & [b] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}\right)$$

В результате вычисления транзитивного замыкания появилось большое количество новых рёбер, однако нас будут интересновать только те, информация о которых храниться в левом верхнем блоке. Остальные рёбра не соответствуют принимающим путям в рекурсивном автомате (убедитесь в этом самостоятельно).

После добавления соответствующих рёбер, мы получим следующий граф:



Его матрица смежности:

$$M_2 = \begin{pmatrix}
[S] & [a] & \mathbf{[S]} & \cdot & \mathbf{[S]} & \cdot & \mathbf{[S]} \\
\cdot & [S] & [b] & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & [S] & [a] & \mathbf{[S]} & \cdot & \mathbf{[S]} \\
\cdot & \cdot & \cdot & [S] & [b] & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & [S] & [a] & \mathbf{[S]} \\
\cdot & \cdot & \cdot & \cdot & \cdot & [S] & [b] \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S]
\end{pmatrix}$$

Таким образом видно, что для выбранных входных данных алгоритму достаточно двух итераций основного цикла: первая содержательная, вторая, как обычно, проверочная. Читателю предлагается самостоятельно выяснить, сколько умножений матриц потребуется, чтобы вычислить транзитивное замыкание на первой итерации.

Теперь запустим алгоритм на второй грамматике и том же входе.

$$M_3 = M_1^1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . & . \\ . & . & [S] & . & . \\ . & . & . & [b] & . \\ . & . & . & . & [S] \\ . & . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} [S] & [a] & . & . & . & . & . \\ . & [S] & [b] & . & . & . & . \\ . & . & [S] & [a] & . & . & . \\ . & . & . & [S] & [b] & . & . \\ . & . & . & . & [S] & [a] & . \\ . & . & . & . & . & [S] & [b] \\ . & . & . & . & . & . & [S] \end{pmatrix} =$$

$$= \left(\begin{array}{ccccccc|ccccccc|ccccccc|ccccccc|ccccccc}
. & . & . & . & . & . & . & . & [a] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & [a] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & [a] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] \\
\hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right)$$

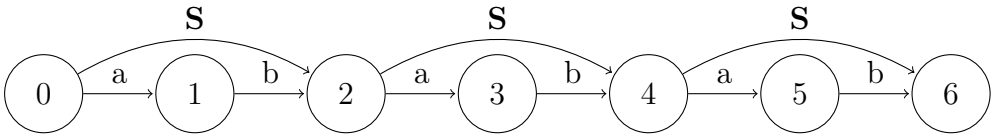
Уже сейчас можно заметить, что размер матриц, с которыми нам придётся работать, существенно увеличился, по сравнению с предыдущим вариантом.

Это, конечно, закономерно, ведь в рекурсивном автомате для предыдущего варианта меньше состояний, а значит и матрица смежности имеет меньший размер.

Транзитивное замыкание:

$$tc(M_3) = \left(\begin{array}{c|c|c|c}
\begin{array}{cccccccc}
[a] & \texttt{[aS]} & & & & \texttt{[aSb]} & & \texttt{[aSbS]} \\
 & & & & & & & \\
[a] & & \texttt{[aS]} & & & \texttt{[aSb]} & & \texttt{[aSbS]} \\
 & & & & & & & \\
[a] & & & \texttt{[aS]} & & \texttt{[aSb]} & & \texttt{[aSbS]} \\
 & & & & & & &
\end{array} &
\begin{array}{cccccccc}
 & & & & & & & \\
[S] & & & & & & & \\
 & [S] & & & & & & \\
 & & [S] & & & & & \\
 & & & [S] & & & & \\
 & & & & [S] & & & \\
 & & & & & [S] & & \\
 & & & & & & [S] & \\
 & & & & & & & \\
 & & & & & [b] & & \\
 & & & & [b] & & & \\
 & & & & & & [b] & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & &
\end{array} &
\begin{array}{cccccccc}
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
[S] & & & & & & & \\
 & [S] & & & & & & \\
 & & [S] & & & & & \\
 & & & [S] & & & & \\
 & & & & [S] & & & \\
 & & & & & [S] & & \\
 & & & & & & [S] & \\
 & & & & & & & [S] \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & &
\end{array} &
\begin{array}{cccccccc}
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & & \\
 & & & & & & &
\end{array}
\end{array}\right)$$

Обновлённый граф:



Его матрица смежности:

$$M_2 = \begin{pmatrix}
[S]\,[a]\,\texttt{[S]} & . & . & . & . \\
. \; [S]\,[b] & . & . & . & . \\
. & . \; [S]\,[a]\,\texttt{[S]} & . & . & \\
. & . & [S]\,[b] & . & . \\
. & . & . & [S]\,[a]\,\texttt{[S]} & \\
. & . & . & . \; [S]\,[b] & \\
. & . & . & . & [S]
\end{pmatrix}$$

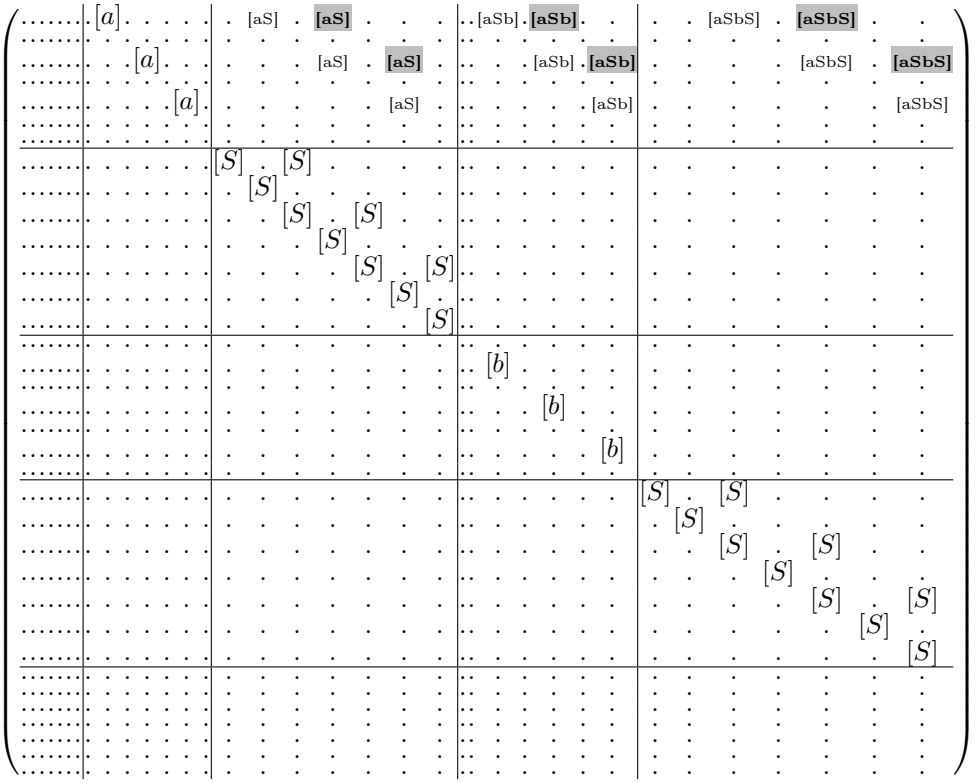
Потребуется ещё одна итерация.

$$M_3 = M_1^1 \otimes M_2 = \begin{pmatrix} \cdot & [a] & \cdot & \cdot & \cdot \\ \cdot & \cdot & [S] & \cdot & \cdot \\ \cdot & \cdot & \cdot & [b] & \cdot \\ \cdot & \cdot & \cdot & \cdot & [S] \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} [S] & [a] & [S] & \cdot & \cdot & \cdot & \cdot \\ \cdot & [S] & [b] & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & [S] & [a] & [S] & \cdot & \cdot \\ \cdot & \cdot & \cdot & [S] & [b] & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & [S] & [a] & [S] \\ \cdot & \cdot & \cdot & \cdot & \cdot & [S] & [b] \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [S] \end{pmatrix} =$$

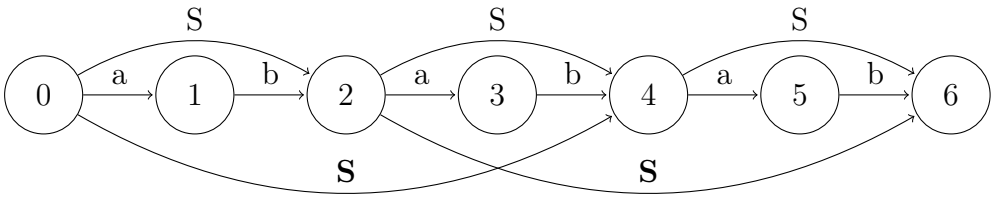
$$= \left(\begin{array}{c|c|c|c} \cdots & [a] \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & [a] \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & [a] \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \hline \cdots & [S] \; \mathbf{[S]} \cdots & \cdots & \cdots \\ \cdots & [S] \cdots & \cdots & \cdots \\ \cdots & [S] \; \mathbf{[S]} \cdots & \cdots & \cdots \\ \cdots & [S] \cdots & \cdots & \cdots \\ \cdots & [S] \; \mathbf{[S]} \cdots & \cdots & \cdots \\ \cdots & [S] \cdots & \cdots & \cdots \\ \cdots & [S] \cdots & \cdots & \cdots \\ \hline \cdots & \cdots & [b] \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & [b] \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & [b] \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \hline \cdots & \cdots & [S] \; \mathbf{[S]} \cdots & \cdots \\ \cdots & \cdots & [S] \cdots & \cdots \\ \cdots & \cdots & [S] \; \mathbf{[S]} \cdots & \cdots \\ \cdots & \cdots & [S] \cdots & \cdots \\ \cdots & \cdots & [S] \; \mathbf{[S]} \cdots & \cdots \\ \cdots & \cdots & [S] \cdots & \cdots \\ \cdots & \cdots & [S] \cdots & \cdots \\ \hline \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \end{array}\right)$$

Транзитивное замыкание:

$tc(M_3) =$



Обновлённый граф:



На этом шаге мы смогли “склеить” из подстрок, выводимых из S, более длинные пути. Однако, согласно правилам грамматики, мы смогли “склеить” только две подстроки в единое целое.

Матрица смежности обновлённого графа:

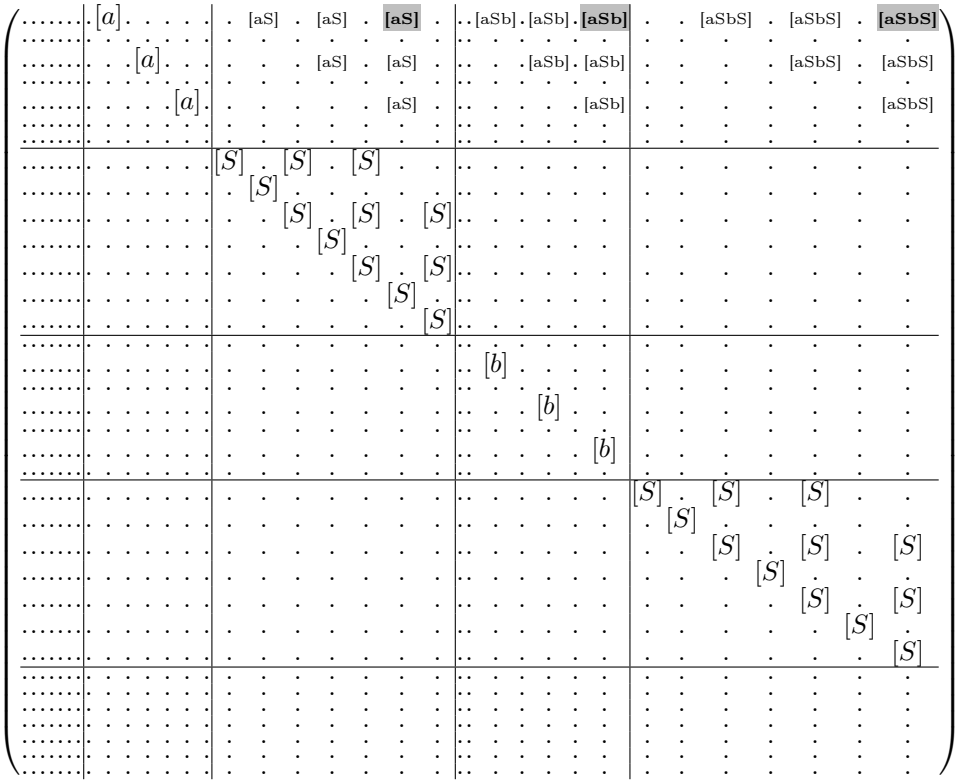
$$M_2 = \begin{pmatrix} [S] & [a] & [S] & . & \mathbf{[S]} & . & . \\ . & [S] & [b] & . & . & . & . \\ . & . & [S] & [a] & [S] & . & \mathbf{[S]} \\ . & . & . & [S] & [b] & . & . \\ . & . & . & . & [S] & [a] & [S] \\ . & . & . & . & . & [S] & [b] \\ . & . & . & . & . & . & [S] \end{pmatrix}$$

И, наконец, последняя содержательная итерация.

$$M_3 = M_1^1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . & . \\ . & . & [S] & . & . \\ . & . & . & [b] & . \\ . & . & . & . & [S] \\ . & . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} [S] & [a] & [S] & . & [S] & . & . \\ . & [S] & [b] & . & . & . & . \\ . & . & [S] & [a] & [S] & . & [S] \\ . & . & . & [S] & [b] & . & . \\ . & . & . & . & [S] & [a] & [S] \\ . & . & . & . & . & [S] & [b] \\ . & . & . & . & . & . & [S] \end{pmatrix} =$$

$$= \left(\begin{array}{ccccccc|ccccccc|ccccccc|ccccccc|ccccccc}
. & . & . & . & . & . & . & . & [a] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & [a] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & [a] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ \hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & [S] & . & \mathbf{[S]} & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & [S] & . & \mathbf{[S]} & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ \hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [b] & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\ \hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & [S] & . & \mathbf{[S]} & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & [S] & . & \mathbf{[S]} \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . & [S] \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & [S] \\ \hline
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . \\
. & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & . & .
\end{array}\right)$$

Транзитивное замыкание:

$tc(M_3) =$ 

В конечном итоге мы получаем такой же результат, как и при первом запуске. Однако нам потребовалось выполнить существенно больше итераций внешнего цикла, а именно четыре (три содержательных и одна проверочная). При этом, в ходе работы нам пришлось оперировать существенно бóльшими матрицами: 35×35 против 21×21.

Таким образом, видно, что минимизация представления запроса, в частности, минимизация рекурсивного автомата как конечного автомата над смешанным алфавитом может улучшить производительность выполнения запросов.

6.5 Особенности реализации

Как и алгоритмы, представленные в разделе 5, представленный здесь алгоритм оперирует разреженными матрицами, поэтому, к нему применимы все те же соображения, что и к алгоритмам, основанным на произведении матриц. Более того, так как результат тензорного произведения является блочной матрицей, то могут оказаться полезными различные форматы для хране-

ния блочно-разреженных матриц. Вместе с этим, в некоторых случаях матрицу смежности рекурсивного автомата удобнее представлять в классическом, плотном, виде, так как для некоторых запросов её размер мал и накладные расходы на представление в разреженном формате и работе с ним будут больше, чем выигрыш от его использования.

Также заметим, что блочная структура матриц даёт хорошую основу для распределённого умножения матриц при построении транзитивного замыкания.

Вместо того, чтобы перезаписывать каждый раз матрицу смежности входного графа M_2 можно вычислять только разницу с предыдущим шагом. Для этого, правда, потребуется хранить в памяти ещё одну матрицу. Поэтому нужно проверять, что вычислительно дешевле: поддерживать разницу и потом каждый раз поэлементно складывать две матрицы или каждый раз вычислять полностью произведение.

Заметим, что для решения задачи достижимости нам не нужно накапливать пути вдоль рёбер, как мы это делали в примерах, соответственно, во-первых, можно переопределить тензорное произведение так, чтобы его результатом являлась булева матрица, во-вторых, как следствие первого изменеия, транзитивное замыкание для булевой матрицы можно искать с применением соответствующих оптимизаций.

6.6 Вопросы и задачи

1. Оценить пространсвенную сложность алгоритма.
2. Оценить временную сложность алгоритма.
3. Найти библиотеку для тензорного произведения. Реализовать алгоритм. Можно предпологать, что запросы содержат ограниченное число терминалов и нетерминалов. Провести замеры. Сравнить с матричным.
4. Реализовать распределённое решение. См. блочную структуру

Глава 7

Сжатое представление леса разбора

Матричный алгоритм даёт нам ответ на вопрос о достижимости, но не предоставляет самих путей. Что делать, если мы хотим построить все пути, удовлетворяющие ограичениям?

Проблема в том, что искомое множество путей может быть бесконечным. Можем ли мы предложить конечную структуру, однозначно описывающую такое множество? Вспомним, что пересечение контекстно-свободного языка с регулярным — это контекстно-свободный язык. Мы знаем, что конекстно-свободный язык можно описать коньекстно-своюодной граммтикой, которая конечна. Это и есть решение нашего вопроса. Осталось толко научиться строить такую грамматику.

Прежде, чем двинуться дальше, рекомендуется вспомнить всё, что касается деревьев вывода 2.2.

7.1 Лес разбора как представление контекстно-свободной грамматики

Для начала нам потребуется внести некоторые изменения в конструкцию дерева вывода.

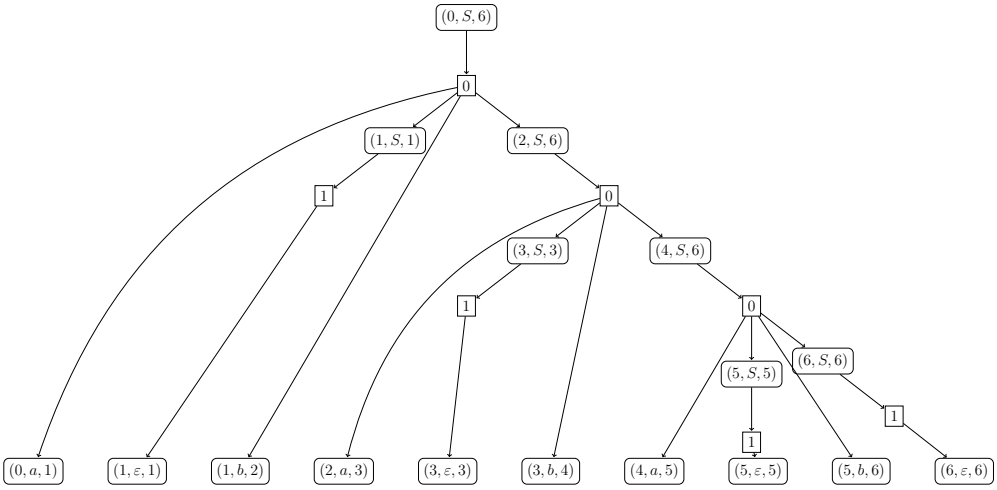
Во-первых, заметим, что в дереве вывода каждый узел соответсвует выводу какой-то подстроки с известными позициями начала и конца. Давайте будем сохранять эту информацию в узлах дерева. Таким образом, метка любого узла это тройка вида (i, q, j), где i — координата начала подстроки, со-

ответствующей этому узлу, j — координата конца, $q \in \Sigma \cup N$ — метка как в исходном определении.

Во-вторых, заметим, что внутренний узел со своими сыновьями связаны с продукцией в грамматике: узел появляется благодаря применению конкретной продукции в процессе вывода. Давайте занумеруем все продукции в грамматике и добавим в дерево вывода ещё один тип узлов (дополнительные узлы), в которых будем хранить номер применённой продукции. Получим следующую конструкцию: непосредственный предок дополнительного узла — это левая часть продукции, а непосредственные сыновья дополнительного узла — это правая часть продукции.

Пример 7.1.1. Построим модифицированное дерево вывода цепочки ${}_0a_1b_2a_3b_4a_5b_6$ в грамматике

$$G_0 = \langle \{a, b\}, \{S\}, S, \{ \\ (0)S \to a\ S\ b\ S, \\ (1)S \to \varepsilon \\ \}\rangle$$



Сохраняемая нами дополнительная информация позволит переиспользовать узлы в том случае, если деревьев вывода оказалось несколько (в случае неоднозначной грамматики). При этом мы можем не бояться, что переиспользование узлов может привести к появлению ранее несуществовавших деревьев вывода, так как дополнительная информация позволяет делать только “безопасные” склейки и затем восстанавливать только корректные деревья. Таким

образом, мы можем представить лес вывода в виде единой структуры данных без дублирования информации.

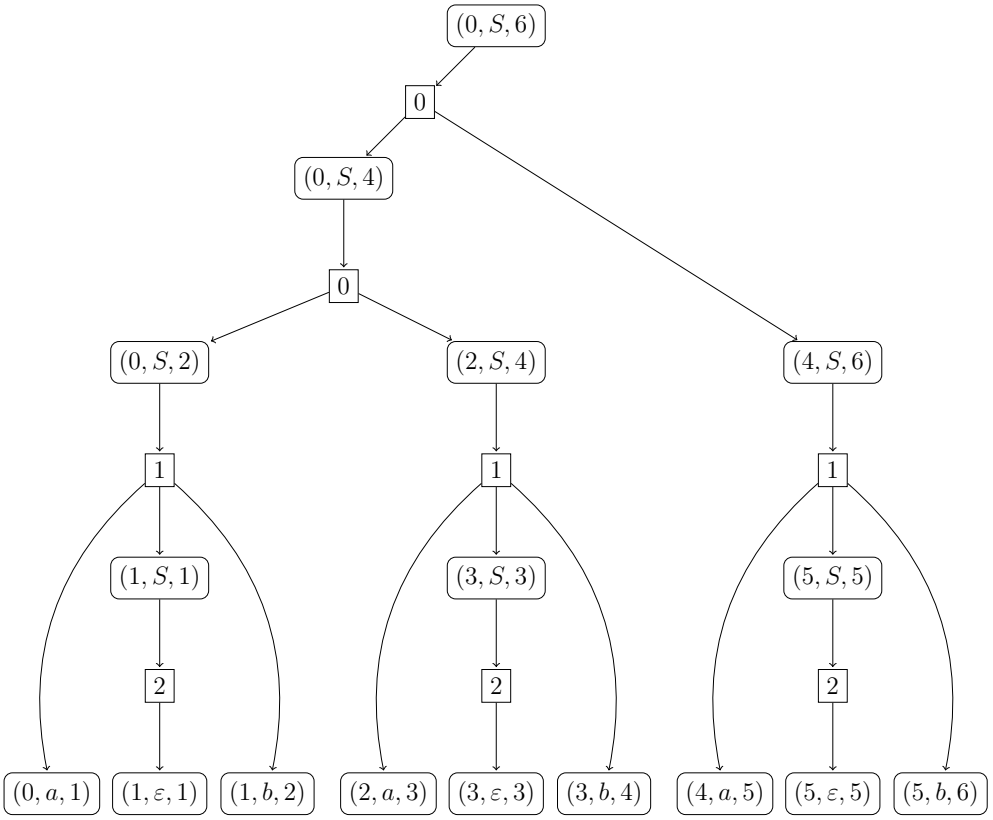
Пример 7.1.2. Сжатие леса вывода. Построим несколько деревьев вывода цепочки ${}_0a_1b_2a_3b_4a_5b_6$ в грамматике

$$G_1 = \langle \{a, b\}, \{S\}, S, \{ \\ (0) S \to SS, \\ (1) S \to a\ S\ b, \\ (2) S \to \varepsilon \\ \}\rangle$$

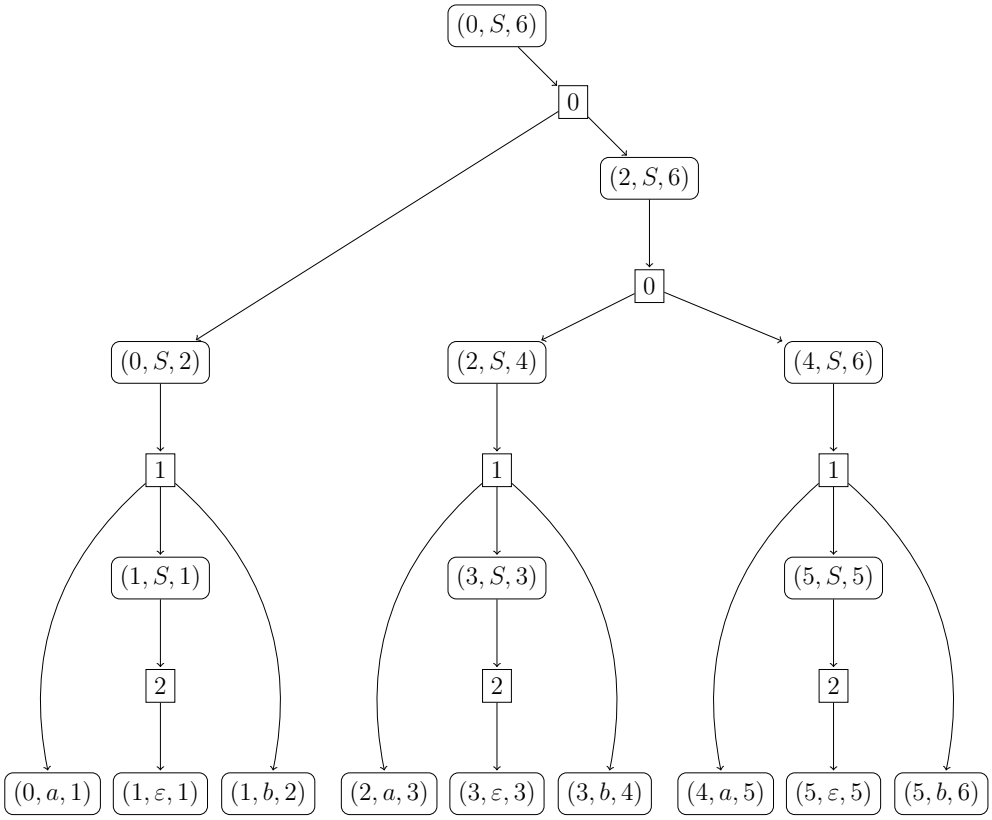
Пердположим, что мы строим левосторонний вывод. Тогда после первого примененеия продукции 0 у нас есть два варианта переписывания первого нетерминала: либо с применением продукции 0, либо с применением продукции 1:

$$\mathbf{S} \xrightarrow{0} \mathbf{S}S \xrightarrow{0} \mathbf{S}SS \xrightarrow{1} a\mathbf{S}bSS \xrightarrow{2} ab\mathbf{S}S \xrightarrow{1} aba\mathbf{S}bS \xrightarrow{2} abab\mathbf{S} \xrightarrow{1} ababa\mathbf{S}b \xrightarrow{2} ababab$$
$$\mathbf{S} \xrightarrow{0} \mathbf{S}S \xrightarrow{1} a\mathbf{S}bS \xrightarrow{2} ab\mathbf{S} \xrightarrow{0} ab\mathbf{S}S \xrightarrow{1} aba\mathbf{S}bS \xrightarrow{2} abab\mathbf{S} \xrightarrow{1} ababa\mathbf{S}b \xrightarrow{2} ababab$$

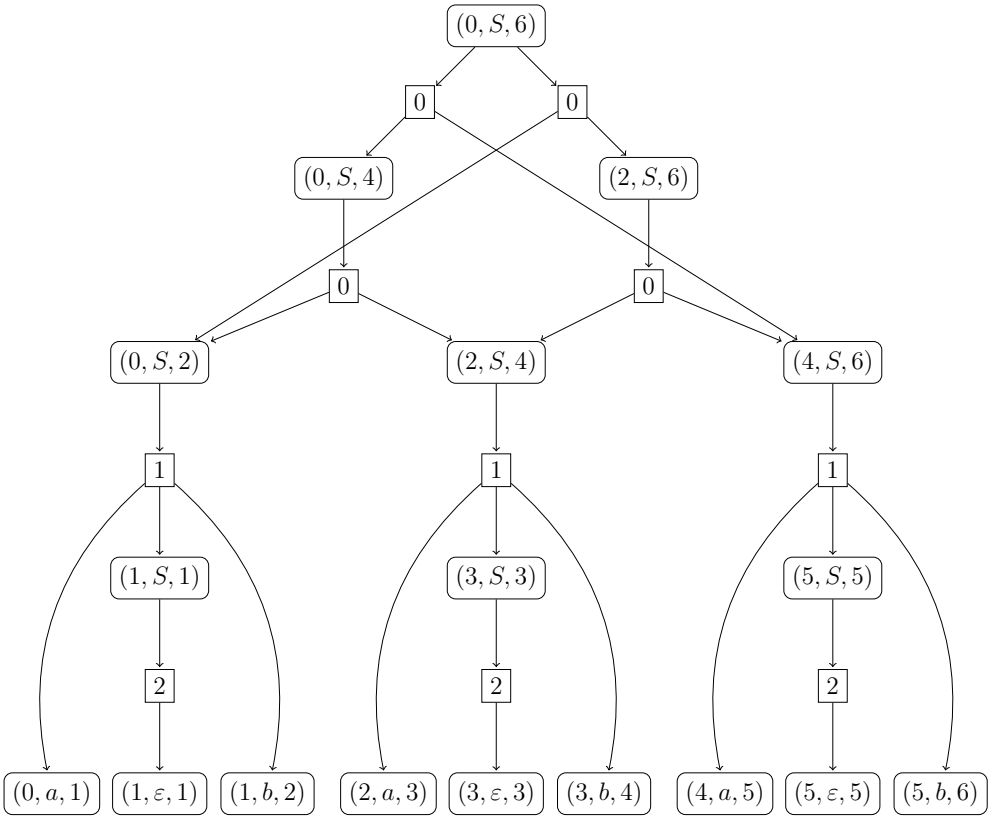
Сначал рассмотрим первый вариант (применили переписываение по подукции 0). Все остальные шаги вывода деретерминированы и в результате мы получим следующее дерево разбора:



Теперь рассмотрим второй вариант — применить продукцию 1. Остальные шаги вывода всё также детерминированы. В результате мы получим следующее дерево вывода:



В двух построенных деревьях большое количество одинаковых узлов. Построим структуру, которая содержит оба дерева и при этом никакие нетерминальные и терминальные узлы не встречаются дважды. В результате мы молучим следующий граф:



Мы получили очень простой вариант сжатого представления леса разбора (Shared Packed Parse Forest, SPPF). Впервые подобная идея была предложена Джоаном Рекерсом в его кандидатской диссертации [62]. В дальнейшем она нашла широкое применеие в обобщённом (generalized) синтаксическом анализе и получила серьёзное развитие. В частности, наш вариант, хоть и позволяет избежать экспоненциального разростания леса разбора, всё же не является оптимальным. Оптимальное асимптотическое поведение достигается при использовании бинаризованного SPPF [13] — в этом случае объём леса составляет $O(n^3)$, где n — это длина входной строки.

Различные модификации SPPF применяются в таких алгоритмах синтаксического анализа, как RNGLR [67], бинаризованная верся SPPF в BRNGLR [70] и GLL [68, 2][1].

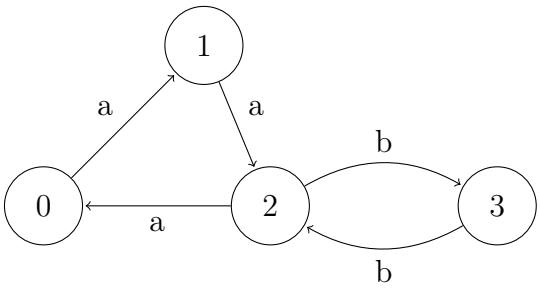
В действительности SPPF может содержать в себе циклы. Для линейного входа их можно получить, когда есть возможность выводить по грамматике бесконечные эпсилон-цепочки. Циклы будут вырожденными, но они будут.

[1]Ещё немного полезной информации про SPPF: `http://www.bramvandersanden.com/post/2014/06/shared-packed-parse-forest/`.

Мы, кроме традиционного использования, будем применять SPPF для представления результатов КС запросов к графам.

В графе может существовать множество способов получить путь из одной вершины в другую. И точно так же при построении деревьев вывода путей может появиться несколько одинаковых нетерминалов, получаемых в разных деревьях по-разному. При объединении в SPPF может оказаться, что какой-то путь из вершины a в вершину b является подпутем другого пути из вершины a в вершину b, просто более длинного. То есть появятся циклические зависимости.

Пример 7.1.3. Рассмотрим пример SPPF для задачи поиска путей с КС ограничениями. Пусть дан граф $\mathcal{G}$:



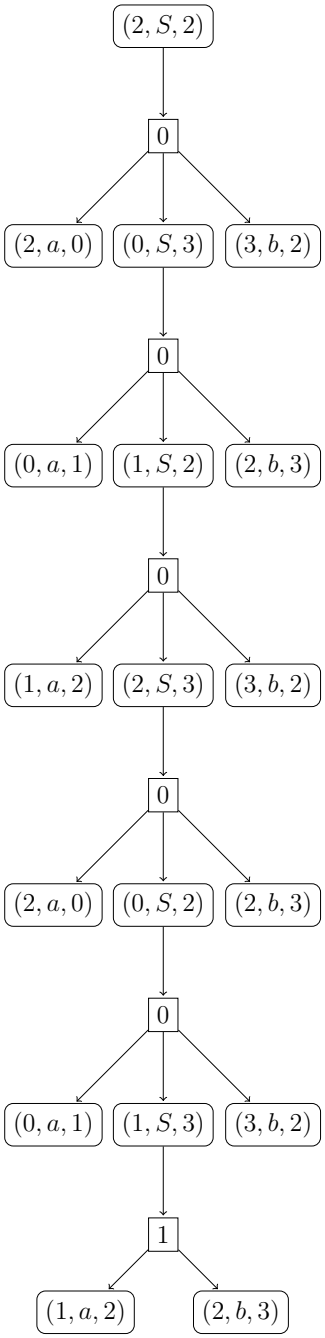
Дана грамматика

$$G = \langle \{a, b\}, \{S\}, S, \{ \\ (0)\ S \to a\ S\ b, \\ (1)\ S \to a\ b, \\ \}\rangle$$

Попробуем найти все пути из вершины 2 в вершину 2, выводимые из нетерминала S. Проверить наличие такого пути можно используя уже известные нам алгоритмы, однако сами пути пока будем строить “методом пристального взгляда”. Найдем один из них. Пусть это будет

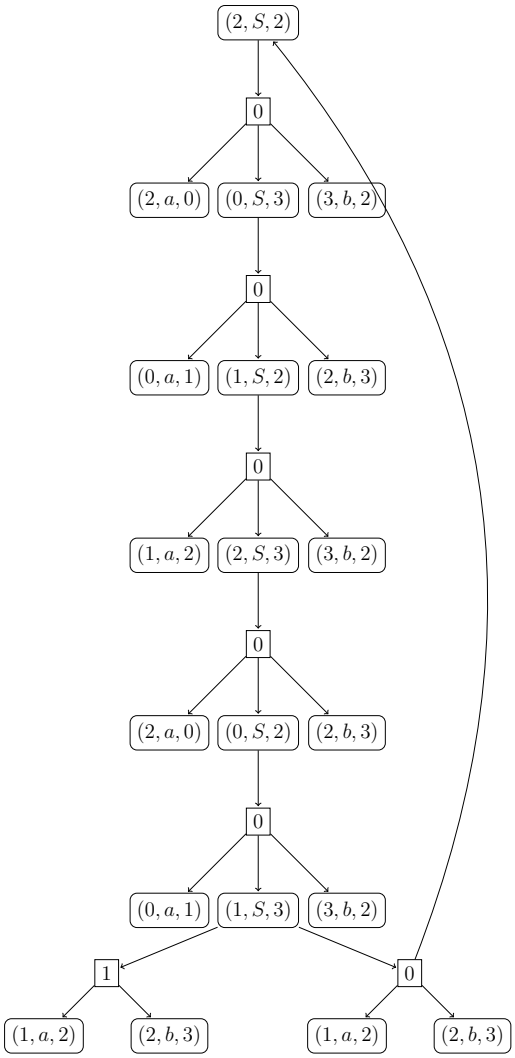
$$2 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{b} 2 \xrightarrow{b} 3 \xrightarrow{b} 2 \xrightarrow{b} 3 \xrightarrow{b} 2.$$

Построим дерево его вывода.



Мы построили дерево вывода для одного пути из вершины 2 в неё же. Но можно заметить, что таких путей бесконечно моного: мы можем бесконечное число раз повтроять уже выполненный обход и получать всё более длинные пути. В терминах дерева вывода это будет означать, что к узлу ${}_1S_3$ мы добавим сына, соответствующего применению продукции 0, а не 1 для нетерминала S.

В таком случае мы получим узел ${}_2S_2$, который уже существует в дереве и таким образом замкнём цикл.



Таким образом мы построили SPPF. Обойдя эту структуру необходимое количество раз, мы можем получить любой путь, удовлетворяющий условию. Более того, в полученном графе можно получать любые другие пути по соответствующим нетерминалам и парам вершин, содержащимся в узлах леса.

Утверждение 7.1.1. SPPF построенный для данной контекстно-свободной грамматки G и графа $\mathcal{G}$

1. содержит терминальный узел вида (i, t_k, j) тогда и только тогда, когда в графе $\mathcal{G}$ есть ребро (i, t_k, j);

2. содержит нетерминальный узел вида (i, S_k, j) тогда и только тогда, когда в графе $\mathcal{G}$ есть путь из вершины i в вершину j, выводимый из нетерминала S_k в грамматике G.

Осталось увидеть, что SPPF является представлением контекстно-свободной грамматики, описывающей результат пересечения исходных графа и грамматики. Для этого просто построим грамматику $G_{SPPF} = \langle \Sigma_{SPPF}, N_{SPPF}, S_{SPPF}, P_{SPPF} \rangle$ по SPPF следующим образом:

- Σ_{SPPF} — все листья SPPF;
- N_{SPPF} — все нетерминальные узлы SPPF;
- S_{SPPF} — нетерминал, соответствующий пути, который нас будет интересовать;
- P_{SPPF} — для каждого дополнительного узла (с номером продукции) добавляем продукцию, левая часть которой — непосредственный предок этого узла, а правая чать — непосредственные потомки.

Пример 7.1.4. Построим грамматику для полученного SPPF:

(0) ${}_2S_2 \to {}_2a_0\ {}_0S_3\ {}_3b_2$
(1) ${}_0S_3 \to {}_0a_1\ {}_1S_2\ {}_2b_3$
(2) ${}_1S_2 \to {}_1a_2\ {}_2S_3\ {}_3b_2$
(3) ${}_2S_3 \to {}_2a_0\ {}_0S_2\ {}_2b_3$
(4) ${}_0S_2 \to {}_0a_1\ {}_1S_3\ {}_3b_2$
(5) ${}_1S_3 \to {}_1a_2\ {}_2S_2\ {}_2b_3$
(6) ${}_1S_3 \to {}_1a_2\ {}_2b_3$

Видим, что для одного единственного нетерминала ${}_1S_3$ существует 2 правила, одно из которых рекурсивное. Попробуем получить левосторонний вывод

какой-нибудь цепочки в этой грамматике:

$$\boldsymbol{{}_2S_2} \overset{(0)}{\Longrightarrow}$$

$${}_2a_0\ \boldsymbol{{}_0S_3}\ {}_3b_2 \overset{(1)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ \boldsymbol{{}_1S_2}\ {}_2b_3\ {}_3b_2 \overset{(2)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ \boldsymbol{{}_2S_3}\ {}_3b_2\ {}_2b_3\ {}_3b_2 \overset{(3)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ \boldsymbol{{}_0S_2}\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2 \overset{(4)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ \boldsymbol{{}_1S_3}\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2 \overset{(5)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ \boldsymbol{{}_2S_2}\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2 \overset{(0)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ \boldsymbol{{}_0S_3}\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2 \overset{(1)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ \boldsymbol{{}_1S_2}\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2 \overset{(2)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ \boldsymbol{{}_2S_3}\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2 \overset{(3)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ \boldsymbol{{}_0S_2}\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2 \overset{(4)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ \boldsymbol{{}_1S_3}\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2 \overset{(6)}{\Longrightarrow}$$

$${}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2a_0\ {}_0a_1\ {}_1a_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2\ {}_2b_3\ {}_3b_2$$

Мы получили цепочку, которая действительно является путем из вершины 2 в вершину 2 в заданном графе. Таким образом выводятся и любые другие соответствующие пути.

7.2 Вопросы и задачи

1. Постройте дерево вывода цепочки $w = aababb$ в грамматике $G = \langle\{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b\ S\}, S\rangle$.

2. Постройте все левосторонние выводы цепочки $w = ababab$ в грамматике $G = \langle\{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b\ |S\ S\}, S\rangle$.

3. Постройте все правосторонние выводы цепочки $w = ababab$ в грамматике $G = \langle\{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b\ |S\ S\}, S\rangle$.

4. Постройте все деревья вывода цепочки $w = ababab$ в грамматике $G = \langle\{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b\ |S\ S\}, S\rangle$, соответствующие левосторонним выводам.

5. Постройте все деревья вывода цепочки $w = ababab$ в грамматике $G = \langle\{a, b\}, \{S\}, \{S \to \varepsilon \mid a\ S\ b \mid S\ S\}, S\rangle$, соответствующие правосторонним выводам.

6. Как связаны между собой леса, полученные в предыдущих двух задачах (4 и 5)? Какие выводы можно сделать из такой связи?

7. Постройте сжатое представление леса разбора, полученного в задаче 4.

8. Постройте сжатое представление леса разбора, полученного в задаче 5.

9. Предъявите контекстно-свободную граммтику существенно неоднозначного языка. Возьмите цепочку длины болше пяти, при надлежащую этому языку, и постройте все деревья вывода этой цепочки в предъявленной граммтике.

10. Постройте сжатое представление леса, полученного в задаче 9.

Глава 8

Алгоритм на основе нисходящего анализа

В данном разделе мы рассмотрим семейство алгоритмов нисходящего синтаксического (рекурсивный спуск, LL, GLL [68, 2]) рассмотрим их обощение для задачи поиска путей с контекстно-свободными ограничениями.

8.1 Рекурсивный спуск

Идея рекурсивного спуска основана на использовании программного стека вызовов в качестве стека магазинного автомата. Достигается это следующим образом.

- Для каждого нетерминала создаётся функция, принимающая ещё не обработанный остаток строки и возвращающая пару: результат вывода префикса данной строки из соответствующего нетерминала и не обработанный остаток строки. В случае распзнователя результат вывода — логическое значение (выводится/не выводится).

- Каждая такая функция реализовывает обработку цепочки согласно правым частям правил для соответствующих нетерминалов: считывание символа входа при обработке терминального символа, вызов соответствующей функции при обработке нетерминального.

У твкого подхода есть два ограничения:

1. Не работает с леворекурсивными грамматиками, грамматиками, в которых вывод может принимать следующий вид:

$$S \to \cdots \to \underline{N_i}\alpha \to \cdots \underline{N_i}\beta \to \cdots \omega$$

2. Шаги должны быть однозначными.

Пример 8.1.1. Постороим функцию рекурсивного спуска для продукции $S \to aSbS \mid \varepsilon$.

Listing 7 Функция рекурсивного спуска

```
1:  function S(ω)
2:      if (len(ω) = 0) then                ▷ Пустая цепочка выводима из S
3:          return (true, ω)
4:      if (ω = a :: tl) then               ▷ Выводимая из S подстрока должна начинаться с a
5:          res, tl' = S(tl)                ▷ Затем должна идти подстрока, выводимая из S
6:          if res && tl' = b :: tl'' then  ▷ Если вызов закончился успешно, то надо
    проверить, что следующий символ — это b
7:              return S(tl'')              ▷ И снова попробовать вывести перфикс из S
8:          else
9:              return (false, tl')
10:     else
11:         return (false, ω)
```

Если возвращаеммое значение этой функции — пара вида *(true, [])*, то разбор завершился успехом.

Данный подход применяется как для ручного написания синтаксических нализаторов, так и при генерации анализаторов по грамматике.

8.2 LL(k)-алгоритм синтаксического анализа

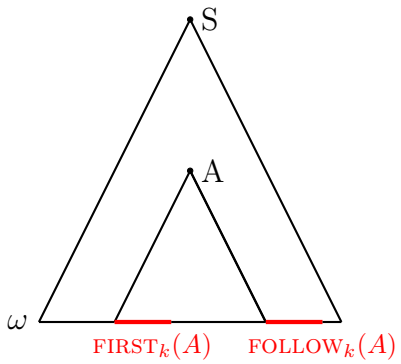
LL(k) — алгоритм синтаксического анализа — нисходящий анализ без отката, но с предпросмотром. Решение о том, какую продукцию применять, принимается на основании k следующих за текущим символом. Временная сложность алгоритма $O(n)$, где n — длина слова.

Алгоритм использует входной буфер, стек для хранения промежуточных данных и таблицу анализатора, которая управляет процессом разбора. В ячейке таблицы указано правило, которое нужно применять, если рассматривается

нетерминал A, а следующие m символов строки — $t_1 \dots t_m$, где $m \leq k$. Также в таблице выделена отдельная колонка для \$ — маркера конца строки.

	...	$t_1 \dots t_m$	...	\$
...	...	...	...	...
A	...	$A \to \alpha$	...	...
...	...	...	...	...

Для построения таблицы вычисляются множества FIRST_k и FOLLOW_k. Идейно их можно понимать, как первые или, соответственно, последующие k символов в результирующем выводе, при использовании нетерминала A. Данную мысль хорошо иллюстрирует рисунок:



Определим их формально:

Определение 8.2.1. Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика. Множество FIRST_k определено для сентециальной формы α следующим образом:

$$\text{FIRST}_k(\alpha) = \{\omega \in \Sigma^* \mid \alpha \overset{*}{\Rightarrow} \omega \text{ и } |\omega| < k \text{ либо } \exists \beta : \alpha \overset{*}{\Rightarrow} \omega\beta \text{ и } |\omega| = k\}$$

, где $\alpha, \beta \in (N \cup \Sigma)^*$.

Определение 8.2.2. Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика. Множество FOLLOW_k определено для сентециальной формы β следующим образом:

$$\text{FOLLOW}_k(\beta) = \{\omega \in \Sigma^* \mid \exists \gamma, \alpha : S \overset{*}{\Rightarrow} \gamma\beta\alpha \text{ и } \omega \in \text{FIRST}_k(\alpha)\}$$

В частном случае для $k = 1$:

$$\text{FIRST}_1(\alpha) = \{a \in \Sigma \mid \exists \gamma \in (N \cup \Sigma)^* : \alpha \overset{*}{\Rightarrow} a\gamma\}, \text{ где } \alpha \in (N \cup \Sigma)^*$$

$\text{FOLLOW}_1(\beta) = \{a \in \Sigma \mid \exists \gamma, \alpha \in (N \cup \Sigma)^* : S \overset{*}{\Rightarrow} \gamma\beta a\alpha\}$, где $\beta \in (N \cup \Sigma)^*$

Множество FIRST_1 можно вычислить, пользуясь следующими соотношениями:

- $\text{FIRST}_1(a\alpha) = \{a\}, a \in \Sigma, \alpha \in (N \cup \Sigma)^*$
- $\text{FIRST}_1(\varepsilon) = \{\varepsilon\}$
- $\text{FIRST}_1(\alpha\beta) = \text{FIRST}_1(\alpha) \cup (\text{FIRST}_1(\beta)$, если $\varepsilon \in \text{FIRST}_1(\alpha))$
- $\text{FIRST}_1(A) = \text{FIRST}_1(\alpha) \cup \text{FIRST}_1(\beta)$, если в грамматике есть правило $A \to \alpha \mid \beta$

Алгоритм для вычисления множества FOLLOW_1:

- Положим $\text{FOLLOW}_1(X) = \varnothing, \forall X \in N$
- $\text{FOLLOW}_1(S) = \text{FOLLOW}_1(S) \cup \{\$\}$, где S— стартовый нетерминал
- Для всех правил вида $A \to \alpha X \beta$: $\text{FOLLOW}_1(X) = \text{FOLLOW}_1(X) \cup (\text{FIRST}_1(\beta) \setminus \{\varepsilon\})$
- Для всех правил вида $A \to \alpha X$ и $A \to \alpha X \beta$, где $\varepsilon \in \text{FIRST}_1(\beta)$: $\text{FOLLOW}_1(X) = \text{FOLLOW}_1(X) \cup \text{FOLLOW}_1(A)$
- Последние два пункта применяются пока есть что добавлять в строящиеся множества.

Пример 8.2.1. Рассмотрим грамматику G со следующими продукциями:

$$\begin{array}{ll} S \to aS' & A' \to b \mid a \\ S' \to AbBS' \mid \varepsilon & B \to c \mid \varepsilon \\ A \to aA' \mid \varepsilon & \end{array}$$

Пример множеств FIRST_1 для нетерминалов грамматики G:

$$\begin{array}{ll} \text{FIRST}_1(S) = \{a\} & \text{FIRST}_1(B) = \{c, \varepsilon\} \\ \text{FIRST}_1(A) = \{a, \varepsilon\} & \text{FIRST}_1(S') = \{a, b, \varepsilon\} \\ \text{FIRST}_1(A') = \{a, b\} & \end{array}$$

Пример множеств FOLLOW_1 для нетерминалов грамматики G:

$$\begin{aligned}
\text{FOLLOW}_1(S) &= \{\$\} \\
\text{FOLLOW}_1(S') &= \{\$\} && (S \to aS') \\
\text{FOLLOW}_1(A) &= \{b\} && (S' \to AbBS') \\
\text{FOLLOW}_1(A') &= \{b\} && (A \to aA') \\
\text{FOLLOW}_1(B) &= \{a, b, \$\} && (S' \to AbBS', \varepsilon \in \text{FIRST}_1(S'))
\end{aligned}$$

Управляющая таблица LL(k) анализатора заполняется следующим образом: продукции $A \to \alpha, \alpha \neq \varepsilon$ помещаются в ячейки (A, a), где $a \in \text{FIRST}_1(\alpha)$, продукции $A \to \alpha$ — в ячейки (A, a), где $a \in \text{FOLLOW}_1(A)$, если $\varepsilon \in \text{FIRST}_1(\alpha)$

Пример 8.2.2. Пример таблицы для грамматики $S \to aSbS \mid \varepsilon$

N	FIRST_1	FOLLOW_1	a	b	\$
S	$\{a, \varepsilon\}$	$\{b, \$\}$	$S \to aSbS$	$S \to \varepsilon$	$S \to \varepsilon$

Однако, не для всех грамматик по множествам FIRST_k и FOLLOW_k возможно выбрать применяемую продукцию, а значит, нельзя однозначно построить таблицу, необходимую для работы алгоритма, поэтому данный алгоритм применим только для грамматик особого класса — LL(k).

Определение 8.2.3. LL(k) грамматика — грамматика, для которой на основании множеств FIRST_k и FOLLOW_k можно однозначно определить, какую продукцию применять.

Важно заметить, что при больших k управляющая таблица сильно разрастается, поэтому на практике данный алгоритм применим для небольших значений k.

Интерпретатор автомата принимает входную строку и построенную управляющую таблицу и работает следующим образом. В каждый момент времени конфигурация автомата это позиция во входной строке и стек. В начальный момент времени стэк пуст, а позиция во входной строке соответствует её началу. На певом шаге в стек добавляются последовательно сперва симаол концы строки, затем стартовый нетерминал. На каждом шаге анализируется существующая конфигурация и совершается одно из действий.

- Если текущая позиция — конец строки и вершина стека — символ конца строки, то успешно завершаем разбор.

- Если текушая вершина стека — терминал, то проверяем, что позиция в строке соответствует этому терминалу. Если да, то снимаем элемент со стека, сдвигаем позицию на единицу и продолжаем разбор. Иначе завершаем разбор с ошибкой.

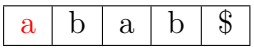
- Если текущая врешина стека — нетерминал N_i и текущий входной символ t_j, то ищем в управляющей таблице ячейку с координатами (N_i, t_j) и записываем на стек содержимое этой ячейки.

Пример 8.2.3. Пример работы LL анализатора. Рассмотрим грамматику $S \to aSbS \mid \varepsilon$ и выводимое слово $\omega = abab$.

Расмотрим пошагово работу алгоритма, будем использовать таблицу, построенную в предыдущем примере:

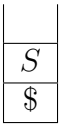
1. Начало работы.

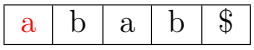
   Стек: 

   входное слово: 

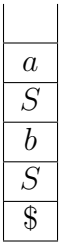
   Финальный символ лежит на стеке, а указатель указывает на первый символ слова.

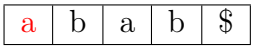
2. кладем стартовый символ на стек

   Стек: 

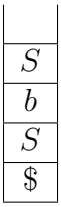
   входное слово: 

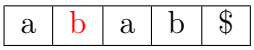
3. Ищем ячейку с координатами (S, a), применяем продукцию из ячейки.

   Стек: 

   входное слово: 

4. Снимаем терминал a со стека и двигаем указатель.

   Стек: 

   входное слово: 

5. Ищем ячейку с координатами (S, b), применяем продукцию из ячейки.

Стек:

b
S
\$

входное слово:

a	b	a	b	\$

6. Снимаем терминал b со стека и двигаем указатель.

Стек:

S
\$

входное слово:

a	b	a	b	\$

7. Ищем ячейку с координатами (S, a), применяем продукцию из ячейки.

Стек:

a
S
b
S
\$

входное слово:

a	b	a	b	\$

8. Снимаем терминал a со стека и двигаем указатель.

Стек:

S
b
S
\$

входное слово:

a	b	a	b	\$

9. Ищем ячейку с координатами (S, b), применяем продукцию из ячейки.

Стек:

b
S
\$

входное слово:

a	b	a	b	\$

10. Снимаем терминал b со стека и двигаем указатель.

Стек:

S
\$

входное слово:

a	b	a	b	\$

11. Ищем ячейку с координатами (S, \$), применяем продукцию из ячейки.

Стек:

\$

входное слово:

a	b	a	b	\$

12. Оказались в конце строки и на вершине стека символ конца — завершаем разбор.

Можно расширить данный алгоритм так, чтобы он строил дерево вывода. Дерево будет строиться сверху вниз, от корня к листьям. Для этого необходимо расширить шаги алгоритма.

- В ситуации, когда мы читаем очередной нетерминал (на вершине стека и во входе одинаковые терминалы), мы создаём лист с соответствующим терминалом.
- В ситуации, когда мы заменяем нетерминал на стеке на правую часть продукции в соответствии с управляющей таблицей, мы создаём нетерминальный узел соответствующий применяемой продукции.

Данное семейство всё так же не работает с леворекурсивными грамматиками и с неоднозначными грамматиками.

Таким образом, по некоторым граммтикам можно построить LL(k) анализатор (назовём их LL(k) граммтиками), но не по всем. С левой рекурсией, конечно, можно бороться, так как существуют алгоритмы устранения левой и скрытой левой рекурсии, а вот с неодносзначностями ничего не поделаешь.

8.3 Алгоритм Generalized LL

Можно построить анализатор, работающий с произвольными КС-грамматиками. Generalized LL (GLL) [68, 2]

Принцип работы остается абсолютно таким же как и для табличного LL:

- Сначала по грамматике строится *управляющая* таблица
- Затем построенная таблица команд и непосредственно анализируемое слово поступают на вход абстрактному интерпретатору.
- Для своей работы интерпретатор поддерживает некоторую вспомогательную структуру данных (стек для LL).
- Один шаг разбора состоит в том, чтобы рассмотреть текущую позицию в слове, применить все соответствующие ей правила из таблицы и при возможности сдвинуть позицию разбора вправо.

Где в этой схеме возникают ограничения на вид обрабатываемой грамматики для алгоритма LL? На самом первом шаге — при построении таблицы может возникнуть ситуация, когда одному нетерминалу N_j и последовательности $first_k(N_j)$ соответствует несколько продукций грамматики. В этом случае грамматика признавалась не соответствующей классу LL(k) и отвергалась анализатором.

Теперь же мы разрешим такую ситуацию и в этом случае в ячейку таблицы будем записывать все продукции грамматики, соответствующие этой ячейке. Однако сразу же возникает вопрос — а что делать интерпретатору, когда при разборе ему необходимо применить правило, состоящее из нескольких продукций? Общий ответ такой — необходим некоторый вид недетерминизма, при котором интерпретатор мог бы "параллельно" обрабатывать несколько возможных вариантов синтаксического разбора.

Эти два свойства (модифицированная управляющая таблица и недетерминизм) суть главные принципиальные отличия GLL(k) от LL(k). Далее мы перейдем к рассмотрению непосредственно технической реализации описанного алгоритма.

Нам необходимо научиться задавать различные ветви (пути) синтаксического разбора и переключаться между ними. Заметим, что состояние любой ветви в любой момент времени суть следующее: необходимо распознать символ $N_j \in N \cup \Sigma$ из продукции X, начиная с элемента слова под индексом i. Т.е. имеем позицию в слове и позицию символа в продукции. Последнее принято называть *слотом грамматики*.

Определение 8.3.1. Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика. *Слотом грамматики* G (позицией грамматики G) назовем пару из продукции $X \in P$ и позиции $0 \leq q \leq length(body(X))$ тела продукции X. При этом введем следующее обозначение $X ::= \alpha \cdot \beta, \quad \alpha, \beta \in (N \cup \Sigma)^*$, где $\cdot$ указывает на позицию в продукции.

Описанная пара позиций уже однозначно задает состояние синтаксического разбора. Имеем множество состояний и переходов между ними — возникает естественное желание воспользоваться терминами графов для представления этой структуры. Такую конструкцию называют *граф-структурированный стек* или *GSS* (Graph Structured Stack), который впервые был предложен Масару Томитой [80] в контексте восходящего анализа. GSS будет являться рабочей структурой нашего нового интерпретатора вместо стека для LL. Состояние разбора вместе с узлом GSS мы будет называть *дескриптором*.

Определение 8.3.2. Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика, X слот грамматики G, i позиция в слове w над алфавитом Σ, а u узел GSS. *Дескриптором* назовём тройку (X, u, i).

Есть несколько способов задания GSS для алгоритма GLL. Вариант, предложенный самими авторами алгоритма, оперирует непосредственно парами из позиции слова и слота грамматики в качестве состояний (и узлов графа) — такой метод является довольно простым и наглядным, но, как описано в работе [2], не самым эффективным. Предложим сразу чуть более оптимальное представление: заметим, что шаги разбора, соответствующие одному и тому же нетерминалу и позиции слова, должны выдавать один и тот же результат независимо от конкретной продукции грамматики, в которой стоит этот нетерминал. Поэтому заводить по узлу на каждый слот грамматики довольно избыточно — вместо этого в качестве состояния будет использовать пары из нетерминала и позиции слова, а позиции грамматики будем записывать на рёбрах.

Итак, мы научились задавать состояния с помощью дескрипторов, а также определились со вспомогательной структурой GSS. Теперь можно перейти к рассмотрению непосредственно самого алгоритма, суть которого довольно проста и напоминает BFS по неявному графу.

Дескриптор задает состояние, которое необходимо обработать. При этом мы без какой-либо дополнительной информации можем продолжить анализ входа из состояния, задаваемого этим дескриптором. В процессе обработки мы можем получить несколько новых состояний. Поэтому будем поддерживать множество R дескрипторов на обработку — на каждом шаге извлекаем один из множества, проводим анализ и кладем в множество новые полученные.

При каких условиях этот процесс будет конечен? Ну, например, если мы каждое состояние будем обрабатывать не более одного раза. И действительно, поскольку наш интерпретатор является “чистым” в том смысле, что для оного и того же состояния каждый раз будут получены одинаковые результаты, проводить анализ дважды не имеет смысла. Поэтому будем также поддерживать множество U всех полученных в ходе разбора дескрипторов, и добавлять в R только те, которых еще нет в U.

И наконец, заключительная и самая главная часть — как происходит обработка дескриптора? Пусть дескриптор имеет вид (X, u, i), а входное слово обозначим W. Есть три возможных варианта, в зависимости от вида позиции грамматики X — разберем каждый из них по отдельности;

- $X ::= \alpha \cdot t\beta$, т.е. указатель смотрит на терминал — в этом случае новых дескрипторов добавлено не будет. Если $W[i] = t$, то мы сдвигаем указатель слота, переходя к рассмотрению $X ::= \alpha t \cdot \beta$, и инкрементируем позицию i в слове. В противном же случае сразу переходим к следующему дескриптору, т.о. терминируя текущую ветвь разбора.

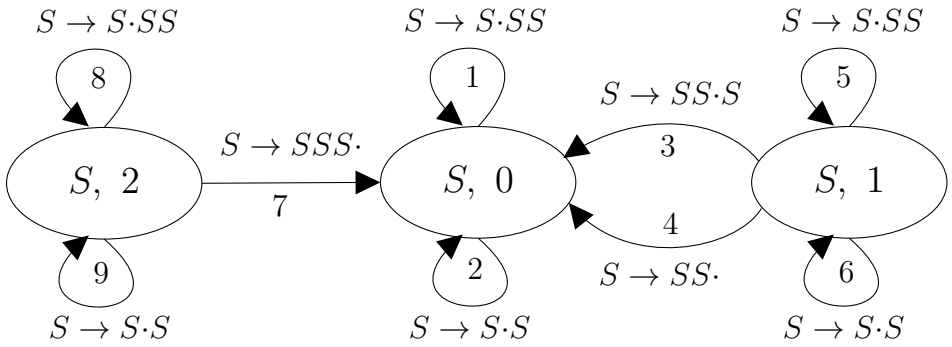
- $X ::= \alpha \cdot A\beta$, т.е. указатель смотрит на нетерминал. Нам нужен GSS узел v вида (A, i) и ребро $(u, X ::= \alpha A \cdot \beta, v)$ (ребро из u в v с пометкой $X ::= \alpha A \cdot \beta$). Если такой узел и ребро уже существуют в нашем GSS, берем их, иначе — создаём. Далее в R добавляем по дескриптору для узла v и каждого правила грамматики из ячейки управляющей таблицы для нетерминала A (конечно, если их еще не было в U). На этом обработка текущего дескриптора завершается.

- $X ::= \alpha\cdot$, т.е. указатель находится в конце продукции. Продукция разобрана, а значит, интерпретатору необходимо вернуться из разбора X к вызывающему правилу и продолжить разбор там (это, в некотором смысле, соответствует возврату из функции разбора нетерминала в методе рекурсивного спуска). По каждому исходящему ребру (u, Y, v) добавляем (если уже не существует) дескриптор (Y, v, i).

Результатом синтаксического разбора является успех тогда и только тогда, когда был достигнут дескриптор вида $(S ::= \alpha\cdot, s, n)$, где слот грамматики представляет собой любое правило для аксиомы S, узел GSS s состоит из аксиомы S и 0, а позиция входного слова равна его длине n. Если же после разбора всех полученных дескрипторов указанный найден не был, результатом будет являться провал.

Давайте посмотрим, как такой алгоритм справится с неоднозачной грамматикой с леворекурсивным правилом.

Пример 8.3.1. Пусть грамматика G имеет вид $S \to SSS \mid SS \mid a$, а разбираемое слово $w = aaa$. Тогда GSS, соответствующий разбору $S \Rightarrow SSS \Rightarrow aSS \Rightarrow aaS \Rightarrow aaa$, будет выглядеть следующим образом (для удобства каждое ребро дополнительно пронумеровано):



Далее мы пошагово рассмотрим процесс его построения, а пока отметим несколько особенностей:

- Это *неполный* GSS. Для задачи синтаксического анализа такого достаточно, поскольку если в какой-то момент был достигнут финальный дескриптор, то обрабатывать все последующие уже не нужно. Однако, для задачи построения SPPF, как мы отметим далее, это уже не так, поскольку она требует агрегирования всех возможных путей разбора.

- Обратите особое внимание на наличие петель. Они как раз-таки и обеспечивают эффективную работу с леворекурсивными правилами, поскольку переиспользуются уже существующие узлы. При этом кратных петель, понятно, не создается, т.к. мы запоминаем все достигнутые дескрипторы в множестве U и дублирующих дескрипторов в рабочее множество R не добавляем.

- В GSS не создаются узлы, соответствующие разбору терминалов (например, $a, 0$). В действительности так можно было бы сделать. Но тогда при обработке слота, указывающего на терминал, сначала бы создался узел GSS, затем интерпретатор сверил бы терминал и символ в слове, после чего, если они совпали, произошел бы возврат из узла, а если нет, узел был бы отброшен и интерпретатор перешел бы к другому дескриптору. Таким образом, при любом случае сначала создается узел, затем выполняется проверка, после чего узел сразу отбрасывается. Для того, чтобы не создавать такие "одноразовые" узлы, проверка терминалов выполняется in-place.

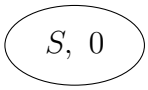
Пронумеруем продукции и выпишем управляющую таблицу:

$S \to SSS \quad (0)$
$S \to SS \quad (1)$
$S \to a \quad (2)$

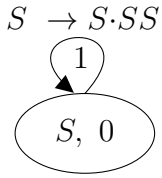
N	FIRST_1	a	\$
S	$\{a\}$	0,1,2	

Разумеется, что конкретный порядок исполнения алгоритма будет зависеть, например, от используемой в качестве рабочего множества R структуры данных и от порядка обработки правил из ячейки управляющей таблицы. Рассмотрим лишь один из возможных вариантов:

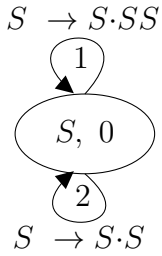
1. Для начала мы создаем узел GSS $s_0 = (S, 0)$ и дескрипторы для правил из ячейки таблицы S, a: $(S \to \cdot SSS, s_0, 0), (S \to \cdot SS, s_0, 0), (S \to \cdot a, s_0, 0)$.



2. При обработке $(S \to \cdot SSS, s_0, 0)$ образовываются петля 1 и дескрипторы $(S \to \cdot SSS, s_0, 0), (S \to \cdot SS, s_0, 0), (S \to \cdot a, s_0, 0)$, которые уже содержатся в множестве U после шага 1 и поэтому не добавляются повторно.

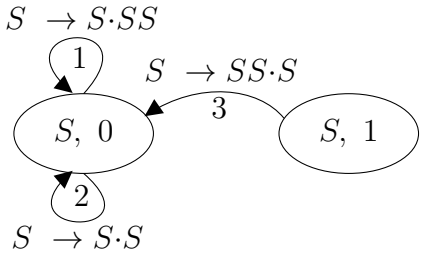


3. При обработке $(S \to \cdot SS, s_0, 0)$ образовывается петля 2, а в остальном аналогично шагу 2.

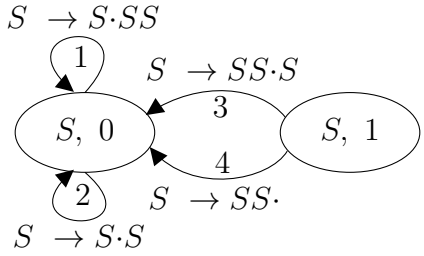


4. При обработке $(S \to \cdot a, s_0, 0)$ мы распознаем терминал a на позиции 0 и, возвращаясь по петлям 1 и 2, добавляем дескрипторы $(S \to S \cdot SS, s_0, 1), (S \to S \cdot S, s_0, 1)$.

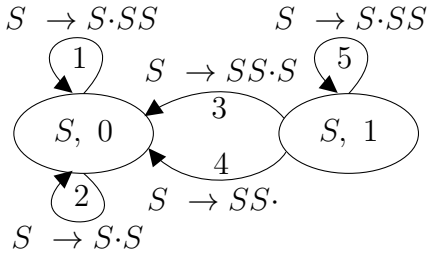
5. При обработке $(S \to S \cdot SS, s_0, 1)$ образовываем узел $s_1 = (S, 1)$ с исходящим ребром 3 и добавляем дескрипторы $(S \to \cdot SSS, s_1, 1), (S \to \cdot SS, s_1, 1), (S \to \cdot a, s_1, 1)$.



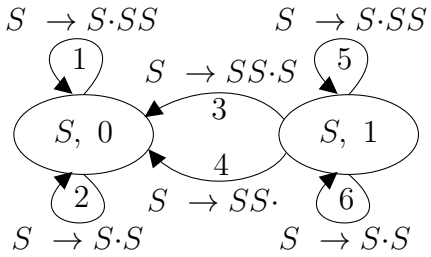
6. При обработке $(S \to S \cdot S, s_0, 1)$ образовываем ребро 4, новых дескрипторов не добавляется.



7. Обработка дескриптора $(S \to \cdot SSS, s_1, 1)$ аналогична шагу 2 с добавлением петли 5.

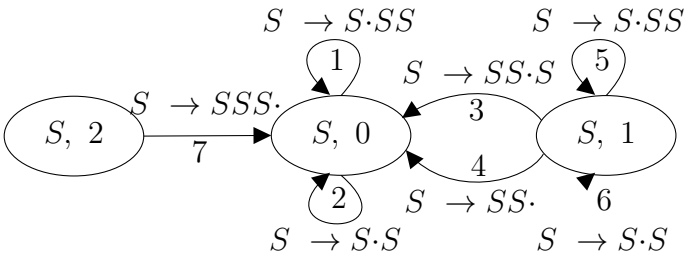


8. Обработка дескриптора $(S \to \cdot SS, s_1, 1)$ аналогична шагу 3 с добавлением петли 6.

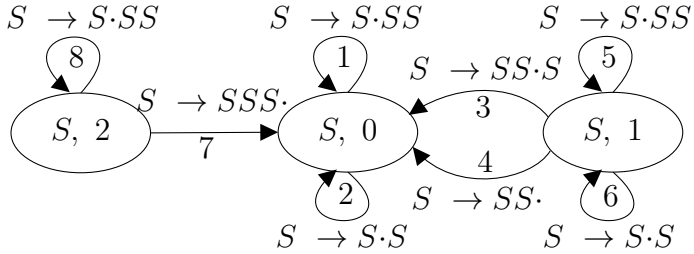


9. При обработке $(S \to \cdot a, s_1, 1)$ мы распознаем терминал a на позиции 1 и, возвращаясь по ребрам 3 и 4, добавляем дескрипторы $(S \to SS \cdot S, s_0, 2), (S \to SS\cdot, s_0, 2)$, а также, возвращаясь по петлям 5 и 6, добавляем дескрипторы $(S \to S \cdot SS, s_1, 2), (S \to S \cdot S, s_1, 2)$.

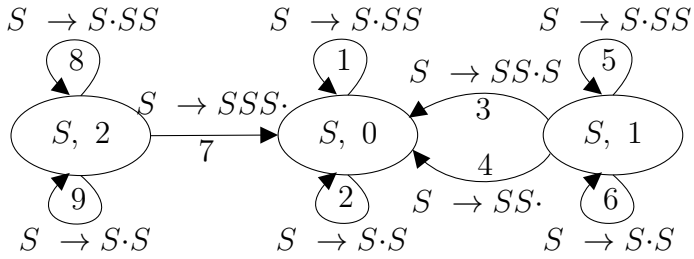
10. При обработке $(S \to SS \cdot S, s_0, 2)$ образовываем узел $s_2 = (S, 2)$ с исходящим ребром 7 и добавляем дескрипторы $(S \to \cdot SSS, s_2, 2), (S \to \cdot SS, s_2, 2), (S \to \cdot a, s_2, 2)$.



11. Обработка дескриптора $(S \to {\cdot}SSS, s_2, 2)$ аналогична шагу 2 с добавлением петли 8.



12. Обработка дескриптора $(S \to {\cdot}SS, s_2, 2)$ аналогична шагу 3 с добавлением петли 9.



13. При обработке $(S \to {\cdot}a, s_2, 2)$ мы распознаем терминал a на позиции 2 и, возвращаясь по ребру 7, добавляем дескриптор $(S \to SSS{\cdot}, s_0, 3)$, а также, возвращаясь по петлям 8 и 9, добавляем дескрипторы $(S \to S \cdot SS, s_2, 3), (S \to S \cdot S, s_2, 3)$.

14. Мы достигли финального дескриптора $(S \to SSS{\cdot}, s_0, 3)$, синтаксический разбор успешен.

Внимательный читатель мог заметить, что если бы в этом примере шаг 4 был выполнен перед шагом 2, разбор довольно быстро бы завершился неудачей. Отсюда вытекает следующее наблюдение: если в какой-то момент из существующего узла появилось новое ребро, необходимо пересчитать все входящие в него пути.

Для построения SPPF требуется внести лишь несколько небольших добавлений:

1. В дескриптор необходимо добавить узел SPPF w, который будет представлять уже разобранный префикс.
2. Необходимо поддерживать множество P из элементов вида (u, z), где u это узел GSS, а z соответствующий ему узел SPPF, для того, чтобы переиспользовать результаты разбора, ассоциированные с узлами GSS.
3. При обработке терминала t на позиции i ищется узел вида $(t, i, i + 1)$, либо создается, если такого еще нет.
4. При обработке нетерминала с помощью P ищется или при необходимости создается промежуточный узел вида (X, l, r), где X соответствующий слот грамматики, а l и r узлы SPPF, отвечающие за разбор левой и правой частей слота соответственно.

Конкретные шаги построения SPPF будут зависеть от выбранного для него формата. Описание эффективного бинаризованного SPPF и детали его построения при выполнении GLL представлены в работе [2].

8.4 Алгоритм вычисления КС запросов на основе GLL

GLL довольно естественно обобщается на граф [32]: позициями входа теперь будем считать не индексы линейного слова, а вершины графа. В самом же алгоритме требуется внести лишь два небольших дополнения:

1. Теперь при обработке терминала “следующих” символов может быть несколько — рассматриваем каждый из них отдельно, сдвигаясь по соответствующему ребру. В результате, при одном чтении можем получить несколько новых дескрипторов, но они независимы, потому просто ставим их в рабочее множество R.
2. При обработке нетерминала, аналогично, правила управляющей таблицы применяются для каждого из “следующих” символов в графе. Соответственно новых дескрипторов будет сгенерировано больше, но все они по-прежнему независимы и просто добавляются в рабочее множество R.

Подробное описание алгоритма и псевдокод представлены в работе [32]. Существует ещё одно обобщение нисходящего синтаксического анализа для решения задачи КС достижимости [48], которое предполагает непосредственное обобщение LL(k) алгоритма, что приводит к аналогичному результату, однако теряется связь с некоторыми построениями.

Основанный на нисходящем анализе алгоритма поиска путей с контекстно-свободными ограничениями имеет следующие особенности.

1. Необходимо явно задавать начальную вершину, поэтому хорошо подходит для задач поиска путей с одним источником (single-source) или с небольшим количеством источников. Для поиска путей между всеми парами вершин необходимо явным образом все указать стартовыми.

2. Является направленным сверху вниз — обходит граф последовательно начиная с указанной стартовой вершины и строит вывод, начиная со стартового нетерминала. Как следствие, в отличие от алгоритмов на основе линейной алгебры и Хеллингса, обойдёт только подграф, необходимый для построения ответа. В среднем это меньше, чем весь граф, который обрабанывается другими алгоритмами.

3. Естественным образом строит множество путей в виде сжатого леса разбора.

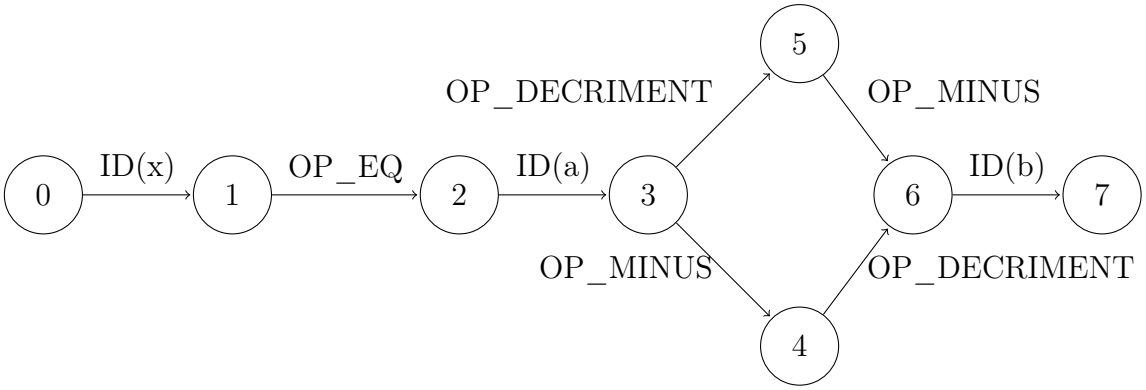
4. Использует существенно более тяжеловесные стуктуры данных и плохо распараллеливается (на практике). Как следствие, при решении задачи досижимости для всех пар путей проигрывает алгоритмам на основе линейной алгебры.

Частным случаем применения задачи КС достижимости является синтаксический анализ с неоднозначной токенизацией, то есть ситуацией, когда несколько пересекающихся подстрок во входной строке символов могут задавать разные лексические единицы и не возможно сделать однозначный выбор на этапе лексического анализа. Например, для строки `x = a---b` возможны несколько вариантов токенизации.

1. `ID(x) OP_EQ ID(a) OP_MINUS OP_DECRIMENT ID(b)`

2. `ID(x) OP_EQ ID(a) OP_DECRIMENT OP_MINUS ID(b)`

В таком случае на вход синтаксическому анализатору можно подать DAG, содержащий все возможные варианты токенизации. Для нашего примера он может выглядить следующим образом:



Далее будем проверять наличие пути из старовой (нулевой) вершины в конечную (соответствующую концу строки). Если таких путей оказалось несколько, то нужны дополнительные средства для выбора нужного дерева разбора. Даннаяя идея рассматривается в работе [69].

Напоследок сделаем небольшое замечание об эффективной реализации: в качестве рабочего множества R можно использовать несколько различных структур данных и, как правило, выбирают очередь. Однако иногда (в особенности для графов) лучше использовать стек дескрипторов, так как в этом случае выше локальность данных — мы кладём пачку дескрипторов, соответствующих исходящим рёбрам. И если граф представлен списком смежности, то исходящие будут храниться рядом и их лучше обработать сразу.

8.5 Вопросы и задачи

1. Проведите алгоритм GLL для грамматики $S \to aSbS \mid \varepsilon$. Правда ли, что эта грамматика принадлежит классу $LL(1)$? Пронаблюдайте, как использование GSS вырождается в работу с обычным стеком.

2. Доразберите все не рассмотренные в примере 8.3.1 дескрипторы, постройте полный GSS.

Глава 9

Алгоритм на основе восходящего анализа

Традиционно, алгоритмы, применяемые для анализа языков программирования как раз умеют строить дерево разбора — то, что нам надо. Только нам бы лес. Вот и посмотрим, как это можно сделать.

Сперва поговорим про классический синтаксический анализ, потом про его адаптацию к анализу графов.

9.1 Восходящий синтаксический анализ

LR(k) — алгоритм восходящего синтаксического анализа. Идея заключается в следующем: входная последовательность символов считывается слева направо с попутным добавлением в стек и выполнением сворачивания — замены последовательности терминалов и нетерминалов, лежащих наверху стека, на нетерминал, если существует соответствующее правило в исходной грамматике.

Определение 9.1.1. Слот (item) — правило грамматики, в правой части которого имеется точка, отделяющая уже разобранную часть правила (слева от точки) от того, что еще предстоит распознать (справа от точки).

Определение 9.1.2. LR-автомат — автомат с магазинной памятью, состояния которого задаются “слотами”, также он имеет следующий набор инструкций:
1. shift p — прочитать следующий символ входной последовательности, положив его в стек, и перейти в состояние p

2. reduce k — применить k-ое правило грамматики, правая часть которого уже лежит на стеке: снимаем правую часть и кладём левую часть

Определение 9.1.3. Предпросмотр — метод, применяемый в синтаксическом анализе. Заключается в следующем: устанавливается максимальное количество входящих символов, которое может быть использовано анализатором для решения того, какое правило использовать (в случае восходящего анализа: к какому правилу нужно свернуться).

Определение 9.1.4. Управляющая таблица — таблица, которая для всех состояний LR-автомата содержит: инструкции для выполнения, если на вершине стека — терминал (при этом в случае LR(k) в каждой ячейке может находиться не более одной инструкции), номер состояния, в которое нужно перейти, если на вершине стека — нетерминал.

Когда в текущем состоянии '.' стоит в конце, мы можем выполнить reduce-инструкцию и перейти в новое состояние. Однако при этом могут возникать следующие конфликты:

- shift-reduce — ситуация, когда не понятно, читать ли следующий символ или выполнить reduce. Например, если правая часть одного из правил является префиксом правой части другого правила: $N \to w, M \to ww'$.
- reduce-reduce — ситуация, когда не понятно, к какому правилу нужно применить reduce. Например, если есть два правила с одинаковыми правыми частями: $N \to w, M \to w$.

Возьмем следующую грамматику:

$$0) S \to aSbS$$
$$1) S \to \varepsilon$$

Расширим вышеупомянутую грамматику, добавив новый стартовый нетерминал S', и далее будем работать с этой расширенной грамматикой:

$$0) S \to aSbS$$
$$1) S \to \varepsilon$$
$$2) S' \to S\$$$

Определение 9.1.5. Замыкание — обобщение понятия "item", заключающееся в добавлении для каждого item'а вида $N \to \alpha.M\beta$ item'ов вида $M \to .\gamma$

Определение 9.1.6. Ядро — исходный item, до применения к нему замыкания.

Пример 9.1.1. Пример ядра и замыкания.

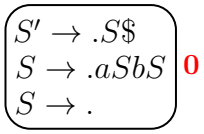
Возьмем правило 2 нашей грамматики, предположим, что мы только начинаем разбирать данное правило.

Ядром в таком случае является item исходного правила: $S' \to .S\$$

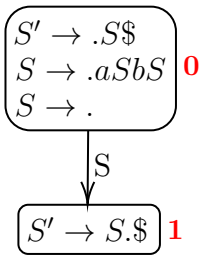
При замыкании добавятся ещё два item'а с правилами по выводу нетерминала 'S', поэтому получаем три item'а: $S' \to .S\$$, $S \to .aSbS$ и $S \to .\varepsilon$

Пример 9.1.2. Пример построения LR-автомата для нашей грамматики с применением замыкания.

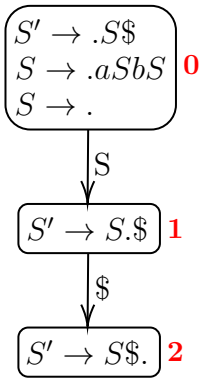
1. Добавляем стартовое состояние: item правила 0 и его замыкание (вместо item'а $S \to .\varepsilon$ будем писать $S \to .$).



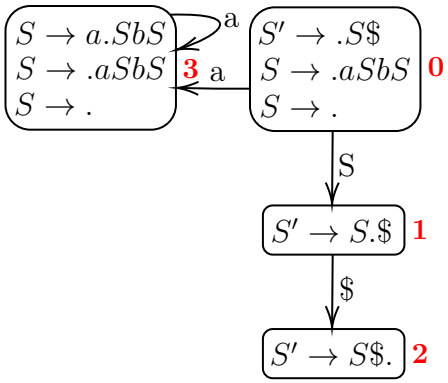
2. По 'S' добавляем переход из стартового состояния в новое состояние 1.



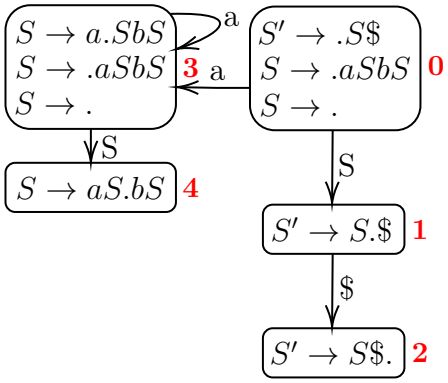
3. По '$' добавляем переход из состояния 1 в новое состояние 2.



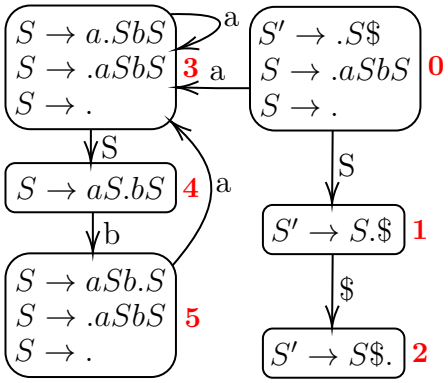
4. По ’a’ добавляем переход из стартового состояния в новое состояние 3 и делаем его замыкание. Также добавляем переход по ’a’ из этого состояния в себя же.



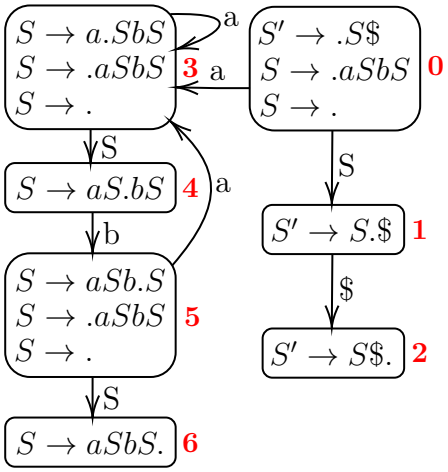
5. По ’S’ добавляем переход из состояния 3 в новое состояние 4.



6. По ’b’ добавляем переход из состояния 4 в новое состояние 5 и делаем его замыкание. Также добавляем переход по ’a’ из этого состояния в состояние 3.



7. По 'S' добавляем переход из состояния 5 в новое состояние 6. Завершаем построение LR-автомата.



Пример 9.1.3. Пример управляющей таблицы для работы с построенным ранее LR-автоматом.

	a	b	$	S
0	shift 3	reduce 1	reduce 1	1
1			ACCEPT	
2				
3	shift 3	reduce 1	reduce 1	4
4		shift 5		
5	shift 3	reduce 1	reduce 1	6
6		reduce 0	reduce 0	

Ход работы LR-парсера. Пусть у нас есть входная строка, LR-автомат со стеком и управляющая таблица.
В начальный момент на стеке лежит стартовое состояние LR-автомата, позиция во входной строке соответствует её началу. На каждом шаге анализируется текущий символ входа и текущее состояние, в котором находится автомат, и совершается одно из действий:

- Если текущая позиция — конец строки и в стеке — стартовый нетерминал исходной грамматики, то успешно завершаем разбор.

- Если в управляющей таблице нет инструкции для текущего состояния автомата и текущего символа на входе, то завершаем разбор с ошибкой.

- Иначе выполняем инструкцию:
  1) в случае shift — кладем на стек текущий символ входа, сдвигая при этом текущую позицию, и номер нового состояния с переходом в него.
  2) в случае reduce — снимаем со стека 2k элементов: k состояний и k терминалов/нетерминалов (где k — длина правой части правила, участвующего в свёртке), кладём на стек нетерминал левой части правила и, оказавшись в некотором состоянии, в котором мы были ранее (самое близкое к вершине из хранимых на стеке состояний), выполняем переход в новое состояние с добавлением его номера в стек, если в управляющей таблице пересечение текущего состояния и добавленного ранее нетерминала — не пусто.

Пример 9.1.4. Пример LR-разбора входного слова abab$ из языка нашей грамматики с использованием построенных ранее LR-автомата и управляющей таблицы.

1. Начало разбора. На стеке — стартовое состояние 0.

   Вход: | a | b | a | b | $ | Стек: | 0 |

2. Выполняем shift 3: сдвигаем указатель на входе, кладем на стек 'a', новое состояние 3 и переходим в него.

   Вход: | a | b | a | b | $ | Стек: | 0 | a | 3 |

3. Выполняем reduce 1 (кладем на стек 'S'), кладем новое состояние 4 и переходим в него.

   Вход: | a | b | a | b | $ | Стек: | 0 | a | 3 | S | 4 |

4. Выполняем shift 5: сдвигаем указатель на входе, кладем на стек 'b', новое состояние 5 и переходим в него.

   Вход: | a | b | a | b | $ | Стек: | 0 | a | 3 | S | 4 | b | 5 |

5. Выполняем shift 3.

   Вход: | a | b | a | b | $ | Стек: | 0 | a | 3 | S | 4 | b | 5 | a | 3 |

6. Выполняем reduce 1, кладем новое состояние 4 и переходим в него.

   Вход: | a | b | a | b | $ | Стек: | 0 | a | 3 | S | 4 | b | 5 | a | 3 | S | 4 |

7. Выполняем shift 5.

   Вход: | a | b | a | b | $ | Стек: | 0 | a | 3 | S | 4 | b | 5 | a | 3 | S | 4 | b | 5 |

8. Выполняем reduce 1, кладем новое состояние 6 и переходим в него.

   Вход: | a | b | a | b | $ | Стек: | 0 | a | 3 | S | 4 | b | 5 | a | 3 | S | 4 | b | 5 | S | 6 |

9. Выполняем reduce 0 (снимаем со стека 8 элементов и кладем 'S'), оказываемся в состоянии 5 и делаем переход в новое состояние 6 с добавлением его на стек.

   Вход: | a | b | a | b | $ | Стек: | 0 | a | 3 | S | 4 | b | 5 | S | 6 |

10. Снова выполняем reduce 0, оказываемся в состоянии 0 и делаем переход в новое состояние 1 с добавлением его на стек. Заканчиваем разбор.

    Вход: | a | b | a | b | $ | Стек: | 0 | S | 1 |

На практике конфликты стараются решать ещё и на этапе генерации. Да, реальные тулы могут сгенерировать парсер по неоднозначной грамматике: из переноса или свёртки выбирать перенос, из нескольких свёрток — первую в каком-то порядке (обычно в порядке появления соответствующих продукций в грамматике).

Существует также модификация LR-разбора, которая называется SLR (Simple LR). Основная идея заключается в том, чтобы хранить в управляющей таблице reduce-инструкции только для тех терминалов, которые встречаются в правилах грамматики сразу после нетерминала, к которому выполняется свертка. Также существует LALR модификация (Look-Ahead LR). В ней применяется склеивание нескольких состояний автомата, входные дуги которых имеют общие символы переходы, в одно состояние. Данные модификации позволяют избежать большее число конфликтов, однако иногда могут иметь таблицы

большего размера, чем в классическом LR. Стоит отметить, что LALR(1)-парсер — менее мощный, чем LR(1)-парсер, но более мощный, чем SLR(1)-парсер.

9.2 GLR и его применение для KC запросов

Алгоритм LR довольно эффективен, однако позволяет работать не со всеми KC-грамматиками, а только с их подмножеством LR(k). Если грамматика находится за рамками допускаемого класса, некоторые ячейки управляющей таблицы могут содержать несколько значений. В этом случае грамматика отвергалась анализатором.

Чтобы допустить множестенные значения в ячейках управляющей таблицы, потребуется некоторый вид недетерминизма, который даст возможность анализатору обрабатывать несколько возможных вариантов синтаксического разбора параллельно. Именно это и предлагает анализатор Generalized LR (GLR) [79]. Далее мы рассмотрим общий принцип работы, проиллюстрируем его с помощью примера, а также рассмотрим модификации GLR.

9.2.1 Классический GLR алгоритм

Впервые GLR парсер был представлен Масару Томитой в 1987 [79]. В целом, алгоритм работы идентичен LR с несколькими принципиальными исключениями:

- Управляющая таблица модифицирована таким образом, чтобы допускать множественные значения в ячейках.
- Для каждой операции reduce, которую мы можем применить на каком-то этапе разбора, создается копия всего стека, после чего, к ней применяется эта операция.
- Если к стеку нельзя применить ни одну операцию shift на следующем входном символе, то этот стек отбрасывается.

Однако, полное копирование стеков приводит к тому, что дублируется слишком много информации. Поэтому можно сделать следующее:

- Объединять одинаковые состояния.

- Объединять одинаковые правые префиксы стеков (то есть верхние их части, к которым быстрее всего доберутся операции reduce).

Для этого стоит использовать более сложную структуру стека: *граф-структурированный стек* или (*GSS*, Graph Structured Stack). Это направленный граф, в котором вершины соответствуют элементам стека, а ребра их соединяют по правилам управляющей таблицы. У каждой вершины может быть несколько входящих и исходящих узлов: таким образом реализуется то самое объединение, упомянутое в предыдущем абзаце.

Пример 9.2.1. Рассмотрим пример GLR разбора с использованием GSS.

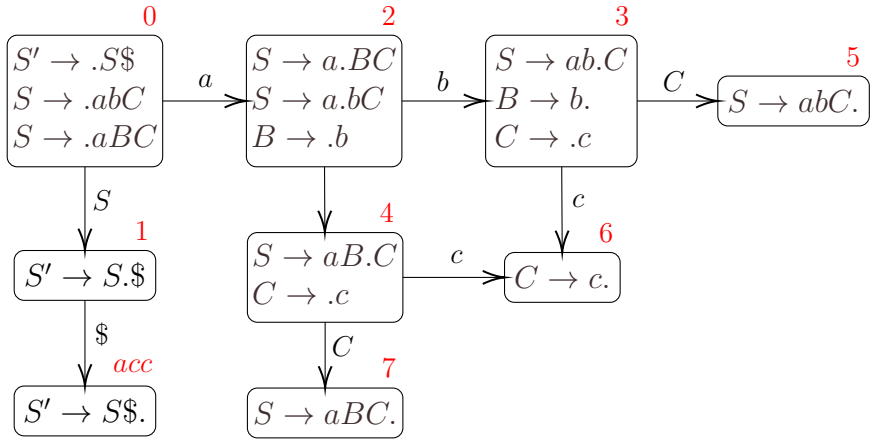
Возьмем грамматику G следующего вида:

0. $S' \to S\$$
1. $S \to abC$
2. $S \to aBC$
3. $B \to b$
4. $C \to c$

Входное слово w:

$$w = abc\$$$

Построим для данной грамматики LR автомат:



И управляющую таблицу:

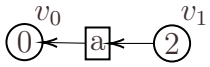
	a	b	c	$	B	C	S
0	shift 2					goto 1	
1				accept			
2		shift 3			goto 4		
3			shift 6 OR reduce 3			goto 5	
4			shift 6			goto 7	
5				reduce 1			
6				reduce 4			
7				reduce 2			

Разберем слово w с помощью алгоритма GLR. Использована следующая аннотация: вершины-состояния обозначены кругами, вершины-символы — прямоугольниками.

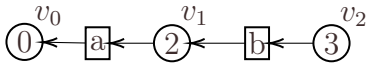
1. Инициализируем GSS стартовым состоянием v_0:

   Вход: | a | b | c | $ | GSS: 

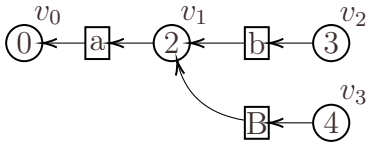
2. Видим входной символ 'a', ищем соответствующую ему операцию в управляющей таблице — $shift\ 2$, строим новый узел v_1:

   Вход: | a | b | c | $ | GSS: 

3. Повторяем для символа 'b', операции $shift\ 3$ и узла v_2:

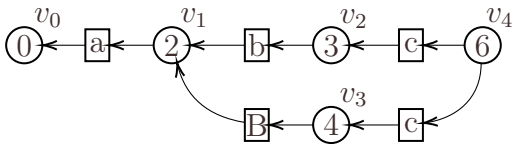
   Вход: | a | b | c | $ | GSS: 

4. При обработке узла v_3 у нас возникает конфликт shift-reduce: $shift\ 6\ OR\ reduce\ 3$. Мы смотрим на вершины, смежные v_2, на управляющую таблицу и на правило вывода под номером 3 для поиска альтернативного построения стека. Находим $goto\ 4$ и строим вершину v_3 с соответствующим переходом по нетерминалу B из v_1 (т.к. количество символов в правой части правила вывода 3 равняется 1, значит мы в дереве опустимся на глубину 1 по вершинам-состояниям):



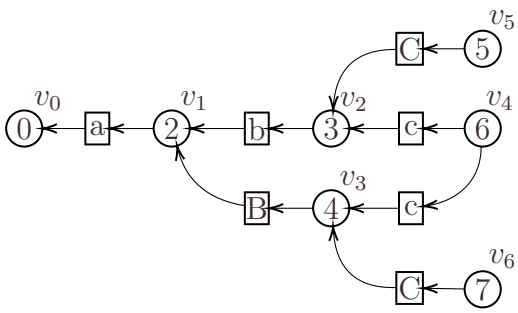
Вход: | a | b | c | $ | GSS:

5. Читаем символ 'c' и ищем в управляющей таблице переходы из состояний 3 и 4 (так как узлы v_2 и v_3 находятся на одном уровне, то есть были построены после чтения одного символа из входного слова). Таким переходом оказывается $shift$ 6 в обоих случаях, поэтому соединяем узел v_4 с обоими рассмотренными узлами:



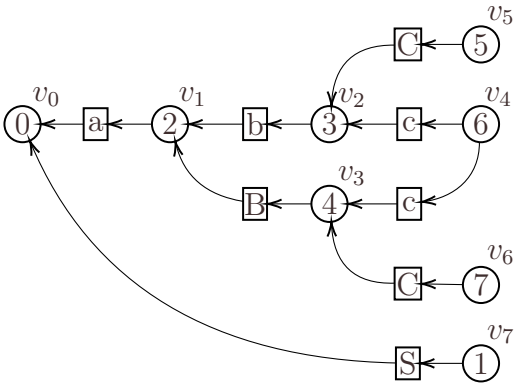
Вход: | a | b | c | $ | GSS:

6. При обработке узла v_4 находим соответствующею 6-ому состоянию редукцию по правилу 4. Его правая часть содержит один символ 'c', 2 вершины-символа с которым достижимы из v_4. Находим вершины-состояния, которые смежны с этими вершинами-символами и обрабатываем переходы по левой части правила 4. Такими переходами по нетерминалу C оказываются $goto$ 5 и $goto$ 7. Строим соответствующие им вершины v_5 и v_6:



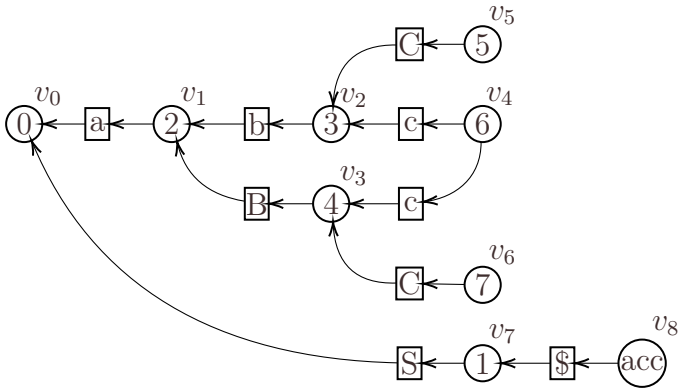
Вход: | a | b | c | $ | GSS:

7. При обработке узлов v_5 и v_6 находим редукции с символом 'S' в левой части и тремя символами в правой. Возвращаемся на 3 вершины-состояния назад и строим вершину v_7 с переходом по S:



Вход: | a | b | c | $ | GSS:

8. Наконец, обрабатывая вершину v_7, читаем символ '$' и строим узел v_8, который соответствует допускающим состоянием:



Вход: | a | b | c | $ | GSS:

9.2.2 Модификации GLR

Алгоритм, представленный Томитой имел большой недостаток: он корректно работал не со всеми КС грамматиками, хоть и расширял класс допустимых LR анализаторами. Объем потребляемой памяти классическим GLR можно оценить как $O(n^3)$ с учетом оптимизаций, о которых говорилось ранее.

Спустя пару лет после публикации Томита-парсера, Элизабет Скотт и Эндриан Джонстоун представили *RNGLR* (Right Nulled GLR) [67] — модифицированная версия GLR, которая решала проблему скрытых рекурсий. Это позволило расширить класс допускаемых грамматик до КС. Однако объем потребляемой памяти можно оценить сверху уже полиномом $O(n^{k+1})$, где k — длина самого длинного правила грамматики, что несколько ухудшило оценку классического GLR.

С этой проблемой справился BRNGLR (Binary RNGLR) [70]. За счет бинаризации удалось получить кубическую оценку сложности и при этом также, как и RNGLR, допускать все КС грамматики.

Кроме того, GLR довольно естественно обобщается до последовательности входных вместо набора . Это происходит следующим образом: элементами во входной структуре теперь будем считать не позиции символа в слове, а вершины графа (то есть "позиция"и множество смежных вершин). Это приводит к тому, что при применении операции shift, следующих символов может быть несколько и каждый из них должен быть рассмотрен отдельно, сдвигаясь по соответствующему ребру и проходя входной граф в ширину. Подробное описание алгоритма и псевдокод представлены в работе [82].

9.3 Вопросы и задачи

1. Постройте LR автомат и управляющую таблицу для грамматики G_1: $S \to aSb$; $S \to \epsilon$.

2. Постройте LR автомат и управляющую таблицу для грамматики G_2: $S \to SSS$; $S \to SS$; $S \to a$.

3. Проведите GLR разбор для грамматики G_2 и входного слова $w = aaa\$$.

4. Реализуйте LR анализатор на любом языке программирования. Программа должна принимать на вход файл с однозначной грамматикой и входное слово, строить LR автомат и управляющую таблицу (во внутреннем представлении), и сообщать, выводимо ли входное слово в данной грамматике.

6*. Реализуйте GLR анализатор на любом языке программирования. Программа должна принимать на вход файл с однозначной грамматикой и входное слово, работать согласно алгоритму GLR и сохранять GSS, а также сообщать, выводимо ли входное слово в данной грамматике.

Глава 10

Комбинаторы для КС апросов

10.1 Парсер комбинаторы

Что это, с чем едят, плюсы, минусы. Про семантику, безопасность, левую рекурсию и т.д. Набор примитивных парсеров и функций, которые умеют из существующих арсеров строить более сложные (собственно, комбинаторы парсеров).

Разобрать символ, разобрать последовательность, разобрать альтернативу. впринципе, этого достаточно, но это не очень удобно.

Проблемы с левой рекурсией. Существуют решения. Одно из них — Meerkat. Подробно про него?

10.2 Комбинаторы для КС запросов

Вообще говоря, идея использовать комбинаторы для навигации по графам достаточно очевидно и не нова. немного про Trails [43].

Комбинаторы для запросов к графам на основе Meerkat [83]

Обобщённые запросы, типобезопасность и всё такое. Примеры запросов.

10.3 Вопросы и задачи

1. Реализовать библиотеку парсер комбинаторов.
2. Что-нибудь полезное с ними сделать.

Глава 11

Производные для КС запросов

В данной главе мы рассмотрим производные Бжозовского и их возможное применение.

11.1 Производные

Впервые производные формальных языков были определены в 1964 году учёным Янушем Бжозовским (в честь него они и были названы). Он определил это понятие для регулярных языков, предложил алгоритм для вычисления производной обобщенного регулярного выражения, также Бжозовский занимался исследованием свойств производных [19].

Изначально всё начиналось с производных регулярных языков, и в главе мы в основном будем говорить именно о них.

Рассмотрим формальное определение производной произвольного языка:

Определение 11.1.1. *Производной языка* $\mathcal{L}$ *по символу* a называется язык $\mathcal{L}' = \partial_a(\mathcal{L}) = \{w \mid aw \in \mathcal{L}\}$, где:

- $\mathcal{L} \subseteq \{w \mid w \in \Sigma^*\}$ — произвольный язык над алфавитом Σ
- $a \in \Sigma$ — символ, по которому берётся производная

То есть фактически производная языка — это суффиксы слов, начинающихся на символ, по которому язык дифференцируется.

Пример 11.1.1 (Производные языка)**.** Рассмотрим язык $\mathcal{L} = \{pen, plain, day, pray\}$ и несколько языков, порождённых от него с помощью дифференцирования:

1. $\mathcal{L}' = \partial_p(\mathcal{L}) = \{en, lain, ray\}$
2. $\mathcal{L}'' = \partial_e(\mathcal{L}') = \{n\}$
3. $\mathcal{L}''' = \partial_n(\mathcal{L}'') = \{\varepsilon\}$

11.2 Принадлежность языку

С помощью производных естественным образом можно проверять принадлежность слова регулярному языку. Есть регулярное выражение R и язык, порожденный этим регулярным выражением $L(R)$. Возьмём произвольное слово w и зададимся вопросом: $w \in L(R)$?

Пусть $w = a_0a_1...a_{n-1}$, а новый язык $\mathcal{L}'$ получен последовательным дифференцирование исходного языка $L(R)$ по каждому из символов слова w, то есть $\mathcal{L}' = \partial_{a_{n-1}}(...\partial_{a_1}(\partial_{a_0}(L(R)))...)$. В таком случае принадлежность w языку $L(R)$ определяется наличием пустого слова в итоговом языке: $\varepsilon \in \mathcal{L}' \Rightarrow w \in L(R)$.

Пример 11.2.1. Дан язык $\mathcal{L} = \{pen, plain, day, pray\}$. Принадлежат ли данному языку слова *pen*, *pet*?

- $\partial_p(\mathcal{L}) = \{en, lain, ray\} \Rightarrow \partial_e(\partial_p(\mathcal{L})) = \{n\} \Rightarrow \partial_n(\partial_e(\partial_p(\mathcal{L}))) = \{\varepsilon\}$. А значит, $pen \in \mathcal{L}$
- $\partial_p(\mathcal{L}) = \{en, lain, ray\} \Rightarrow \partial_e(\partial_p(\mathcal{L})) = \{n\} \Rightarrow \partial_t(\partial_e(\partial_p(\mathcal{L}))) = \emptyset$. Значит, $pet \notin \mathcal{L}$

Описанный выше механизм является основной идеей всех алгоритмом, которые стоятся на производных.

11.3 Построение производных

Концептуально понятно, как выглядят производные, но хотелось бы уметь считать их алгоритмически, причём сохраняя конструктивное представление языка. Например, для КС грамматики, вычисляя её производную, строится другая КС грамматика.

Рассмотрим алгоритм вычисления производной для регулярных языков, которые представлены как регулярные выражения. Пусть r_1 и r_2 — два регулярных выражения, a — произвольный терминальный символ. Определим

вспомогательную функцию N — Nullable, которая проверяет язык на содержание в нём пустого слова ε:

$$\begin{aligned} N(\varepsilon) &= true \\ N(a) &= false \\ N(r_1 \cdot r_2) &= N(r_1) \wedge N(r_2) \\ N(r_1 \mid r_2) &= N(r_1) \vee N(r_2) \\ N(r^*) &= true \end{aligned}$$

Теперь мы готовы перейти непосредственно к вычислению производной по символу a:

$$\begin{aligned} \partial_a(\emptyset) &= \emptyset \\ \partial_a(\varepsilon) &= \emptyset \\ \partial_a(b) &= \begin{cases} \emptyset, & \text{if } a \neq b \\ \varepsilon, & \text{otherwise} \end{cases} \\ \partial_a(r_1 \cdot r_2) &= \begin{cases} \partial_a(r_1) \cdot r_2 \mid \partial_a(r_2) & \text{if } N(r_1) = \text{true} \\ \partial_a(r_1) \cdot r_2, & \text{otherwise} \end{cases} \\ \partial_a(r_1 \mid r_2) &= \partial_a(r_1) \mid \partial_a(r_2) \\ \partial_a(r^*) &= \partial_a(r) \cdot r^* \end{aligned}$$

Все правила, за исключением, может быть, последнего, достаточно тривиальны. Последнее доказывается по индукции, но особо любопытные могут руками вычислить для первой пары слагаемых и рассмотреть закономерность.

Приводить подобный алгоритм для КС языков мы не будем, но важно понимать, что он есть, и идея не сильно отличается от алгоритма для регулярных языков.

11.4 Задача достижимости

Используя производные, можно решать задачу достижимости. Даны регулярный запрос и граф, для простоты зафиксируем стартовую вершину. Для решения задачи достижимости при помощи производных рекурсивно выполняем следующее:

1. При переходе по ребру дифференцируем запрос по метке на нём

2. Передаём эту производную вдоль ребра в следующую вершину

3. В этой следующей вершине производная запоминается, если ранее не встречалось регулярное выражение, которое порождается тот же язык (таким образом формируется набор регулярных выражений, порождающих языки, которые уже передовались на вершину). В противном случае – терминируемся.

После того, как вышеописанный алгоритм завершился для всех вершин, проходим по ним и ищем в их наборах Nullable регулярные выражения, которые и сигнализируют о том, что путь из стартовой веришны в данную вершину существует.

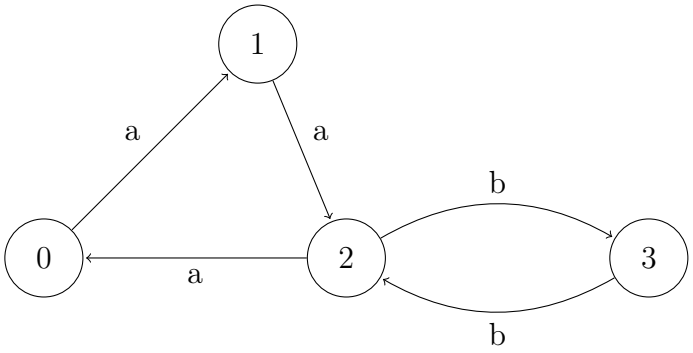
Некоторые замечания:

- Если алгоритм запускается не на одной вершине, а сразу на нескольких, передаются вдоль рёбер не просто производные, а пары: (стартовая вершина, производная).

- Что мы в общем случае делаем с циклами, чтобы задача считалась алгоритмически? Разбиваем граф на наибольшие по включению компоненты сильной связности с связями между ними, стягиваем компоненты. Тогда граф превращается в дек, в котором проблем с нетерминируемостью нет, а с компонентами сильной связности разбираемся отдельно. Для этого достаточно рассмотреть одну такую компоненту: если есть цикл, то по нему можно ходить, только если в запросе есть применение звезды Клини, и, как нам уже известно, производная запроса данной операции — конечная конструкция. В итоге мы сможем, переходя по вершинами, прийти к моменту, когда внутри компоненты в наборах производных вершин новых элементов не будет прибавляться, а значит, мы можем закончить алгоритм.

Рассмотрим решение задачи достижимости с помощью производных на конкретном примере:

Пример 11.4.1. Даны регулярный запрос $R = a^* \mid a^* \cdot b$ и граф $\mathcal{G}$:



Стартовая веришина — вершина графа $\mathcal{G}$ с индексом 0. Хотим проверить существование путей из вершины 0 до других вершин графа по R. Заведем 4 множества, в которых будем хранить регулярные выражения, передаваемые в конкретную вершину: M_0 — для вершины с индексом 0, M_1 — для вершины с индексом 1 и так далее.

1. Начинаем идти вдоль ребра с меткой a из вершины 0 в 1. Дифференцируем наш запрос по данной метке: $\partial_a(R) = \partial_a(a^* \mid a^* \cdot b) = \partial_a(a^*) \mid \partial_a(a^* \cdot b) = \partial_a(a) \cdot a^* \mid \partial_a(a^*) \cdot b \mid \partial_a(b) = a^* \mid a^* \cdot b = R$. Запоминаем эту производную в M_1: $M_1 = \{R\}$

2. Аналогично идём из вершины 1 в вершину 2 по a. В итоге в M_2 также будет находиться R.

3. В вершине 2 параллельно идём по метке a в 0 и по метке b в вершину с индексом 3.

   (a) В первом потоке при переходе по a мы дифференцируем R по a, получаем также R, кладем его в M_0. Пытаясь продолжить эти вычисления в данном потоке, мы обнаружим, что при переходе по a из вершины 0 в вершину 1 наша производная будет снова R, но в множестве M_1 R уже находится, а значит, данная ветка вычислений терминируется.

   (b) При переходе по b диффиринцируем переданную производную R: $\partial_b(R) = \partial_b(a^* \mid a^* \cdot b) = \partial_b(a^*) \mid \partial_b(a^* \cdot b) = \partial_b(a) \cdot a^* \mid \partial_b(a^*) \cdot b \mid \partial_b(b) = \partial_b(b) = \varepsilon$. Запоминаем производную в M_3: $M_3 = \{\varepsilon\}$. Дифференцируем её по b при переходе из 3 в 2. Кладём в M_2 $\partial_b(\varepsilon) = \emptyset$. Таким образом, $M_2 = \{R, \emptyset\}$. На данном этапе можно терминироваться, т.к. для достижимости нам интересны регулярные выражения, порождающий ε, но дифференцируя $\emptyset$ по каким-либо меткам, мы

снова получаем $\emptyset$. При "честном"вычислении, в нашем случае пустое множество будет добавлено во все множества M_i, и уже после этого вычисление завершится.

4. Теперь у нас есть сформированные множества производных для всех вершин и мы можем говорить о достижимости до данных вершин из стартовой. Рассмотрим M_0, $M_0 = \{R\}$. Содержит ли R ε? Посчитаем для этого нашу функцию Nullable: $N(R) = N(a^* \mid a^* \cdot b) = N(a^*) \mid N(a^* \cdot b) = true \mid N(a^* \cdot b) = true$. Из чего следует, что существует путь из вершины 0 в вершину 0. Аналогичным способом заключаем, что веришны 2 и 3 достижимы из вершины 0 (т.к. в их множества также содержится R). Рассмотрим $M_3 = \{\varepsilon\}$. $N(\varepsilon) = true$, значит, вершина 3 тоже достижима из 0. Алгоритм завершён.

Таким образом, выполнив указанный выше алгоритм для данного примера, мы определили, что из стартовой вершины с индексом 0 достижимы все вершины графа $\mathcal{G}$.

11.5 Парсинг на производных

Статьи [49, 1, 50, 4] Реализации. На Scala [1], на Racket [2].

11.6 Адоптация для КС запросов

Для регулярных запросов над графами [52]. Хорошо работают в распределённых системах, в которых реализовван параллелизм уровня вершин. Например Google Pregel.

11.7 Вопросы и задачи

1. Предъявить несколько выводов для одной цепочки.
2. Построить выводы
3. Построить деревья вывода !!! Перенести из раздела про SPPF

[1] `https://github.com/djspiewak/parseback`

[2] `https://bitbucket.org/ucombinator/derp-3/src/86bca8a720231e010a3ad6aefd1aa1c0f35cbf6b/src/derp.rkt?at=master&fileviewer=file-view-default`

Глава 12

От CFPQ к вычислению Datalog-запросов

Рассмотрим грамматику $S \to aSb \mid SS \mid \varepsilon$, заданную через набор предикатов:

- $a(i, w)$ — Предикат, соответствующий терминалу. Обращается в True, если на i-том месте в строке w стоит символ a
- $S(i, j, w)$ — Предикат, соответствующий нетерминалу. Обращается в True, если выполняется одно из условий:
  1. $i == j \quad (\varepsilon)$
  2. $\exists k : i \leq k \leq j \ \ \& \ \ S(i, k-1, w) \ \ \& \ \ S(k, j, w) \quad (SS)$
  3. $a(i, w) \ \ \& \ \ S(i+1, j-1, w) \ \ \& \ \ b(j-1, w) \quad (aSb)$

Таким образом $S(0, |w|, w)$ покажет, выводится ли строка w из данной грамматики, а $S(\_, \_, w)$ даст нам список всех цепочек внутри w, выводящихся из S.

12.0.1 Datalog

Datalog [31][1] — декларативный логический язык программирования. Используется для написания запросов к дедуктивным базам данных[2].

Пример 12.0.1. Пример программы на даталоге.

[1] `https://www.computer.org/csdl/journal/tk/1989/01/k0146/13rRUx0xPIQ`
[2] `https://en.wikipedia.org/wiki/Deductive_database`

1. Набор фактов (часто факты находятся в базе данных):
   - $a(0)$.
   - $b(1)$.
   - $a(2)$.
   - $b(3)$.
   - $s(I, I)$.
2. Набор правил:
   - $s(I, J) :- s(I, K - 1), s(K, J), (I \leq K \leq J)$
   - $s(I, J) :- a(I), s(I + 1, J - 1), b(J)$
3. Запросы:
   - $? - s(I, J)$

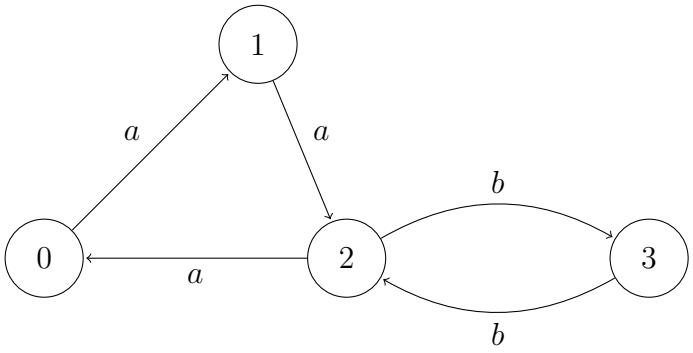
Таким образом мы описали на даталоге строку через набор фактов, грамматику, указанную выше, через набор фактов и правил и сделали запрос на все цепочки, выводящиеся из S.

NB! Обратите внимание [31], что строки, начинающиеся с большой буквы, в даталоге считаются переменными. Также важно, что все переменные неявно квантифицированны.

12.0.2 Datalog для работы с графами

На даталоге также можно задавать графы и писать к ним запросы.

Пример 12.0.2. Пример описания графа на даталоге.



$a(0,1).$

$a(1,2).$

$a(2,0).$

$b(2,3).$

$b(3,2).$

Теперь зададим рассмотренную выше грамматику для работы с графом.

$s(I,I).$

$s(I,J) :- s(I,K-1), s(K,J).$

$s(I,J) :- a(I,L), S(L,M), b(M,J).$

Тогда запрос $? - s(I,J)$ выдаст нам все такие пути в графе, что последовательность составляющих их вершин в порядке прохождения выводится их грамматики.

12.0.3 Алгоритм Эрли

Для распознавания контекстно-свободных грамматик может использоваться алгоритм Эрли [75] [3]. Рассмотрим грамматику $G = (N, T, P, S)$, слово $a_1...a_n$, и правило $A \to \alpha\beta$. Будем считать, что утверждение $[A \to \alpha \bullet \beta](i,j), j \in [1..n]$ является истиной, если верно, что:

- $\alpha \xrightarrow{*} a_{i+1}...a_j$ (Последовательность выводится из α)
- $S \xrightarrow{*} a_1...a_j A_\gamma$

Рассмотрим правила вывода для подобных утверждений:

1. $\frac{S \to \alpha \in P}{[S \to \bullet\alpha](0,0)}$ Инициализация (Init)
2. $\frac{[A \to \alpha \bullet a_{j+1}\beta](i,j)}{[A \to \alpha a_{j+1} \bullet \beta](i,j+1)}$ Сканирование (Scan)
3. $\frac{[A \to \alpha \cdot B\beta](i,j) \quad B \to \gamma \in P}{[B \to \bullet\gamma](j,j)}$ Предсказание (Predict)

[3] `https://en.wikipedia.org/wiki/Earley`

4. $\frac{[A \to \alpha \cdot B](i,j) \quad [B \to \gamma \bullet](j,k)}{[A \to \alpha B \bullet \beta](i,k)}$ Завершение (Complete)

Идея алгоритма Эрли заключается в том, чтобы, начиная с инициализации, используя правила, вывести утверждение, содержащие данную строку слева от точки, и ничего справа, или попробовать все возможные выводы и признать, что строка не выводима.

Сложность алгоритма Эрли составляет $O(|P|^2 n^3)$

Пример 12.0.3. Пример начала одной из веток дерева вывода для алгоритма Эрли для рассматриваемой грамматики

$$\underline{S \to SS} \quad Init$$

$$\underline{[S \to \bullet SS](0,0), S \to aSb} \quad Predict$$

$$\underline{[S \to \bullet aSb](0,0)} \quad Scan$$

$$\underline{[S \to a \bullet Sb](0,1), S \to \varepsilon} \quad Predict$$

$$\underline{[S \to \bullet](0,1), [S \to aS \bullet b](0,1)} \quad Complete$$

$$\underline{[S \to aS \bullet b](0,1)} \quad Scan$$

$$\underline{[S \to \bullet SS](0,0), [S \to aSb\bullet](0,2)} \quad Complete$$

$$\underline{[S \to S \bullet S](0,2)}$$

$$\ldots$$

Сложность можно понизить, изменив правила “Предсказание” и “Завершение” таким образом:

- $\frac{[A \to \alpha \cdot B\beta](i,j) \quad B \to \gamma \in P}{[B \to \bullet\gamma](j,j)} \Rightarrow \frac{[A \to \alpha \cdot B\beta](i,j)}{?B(j)}; \quad \frac{?B(j) \quad B \to \gamma \in P}{[B \to \cdot\gamma](j,j)}$
- $\frac{[A \to \alpha \cdot B](i,j) \quad [B \to \gamma \bullet](j,k)}{[A \to \alpha B \bullet \beta](i,k)} \Rightarrow \frac{[B \to \gamma \bullet](j,k)}{B(j,k)}; \quad \frac{[A \to \alpha \cdot B\beta](i,j) \quad B(j,k)}{[A \to \alpha B \cdot \beta](i,k)}$

Так, разложив каждое правило на два, мы избавляемся от необходимости перевычислять дерево разбора каждого нетерминала после того, как однократно вычислим, что он выводим (мемоизируем его). Получаем сложность $O(|P|)$.

Описанный подход мемоизации [9] части используется для оптимизации программ на даталоге. Можно либо видоизменять правила, задающие грамматику в тексте программы, либо модифицировать компилятор, чтобы он пытался сделать это автоматически.

12.1 Вопросы и задачи

1. Написать синтаксический анализатор раз.
2. Написать синтаксический анализатор два.
3. Побаловаться с неоднозначными грамматиками
4. Побаловаться с конъюнктивными грамматиками.
5. Графы?

Литература

[1] M. D. Adams, C. Hollenbeck, and M. Might. On the complexity and performance of parsing with derivatives. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 224–236, New York, NY, USA, 2016. ACM.

[2] A. Afroozeh and A. Izmaylova. Faster, practical gll parsing. In B. Franke, editor, *Compiler Construction*, pages 89–108, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[3] M. R. Albrecht, G. V. Bard, and W. Hart. Efficient multiplication of dense matrices over GF(2). *CoRR*, abs/0811.1714, 2008.

[4] L. Andersen. Parsing with derivatives.

[5] J. W. Anderson, P. Tataru, J. Staines, J. Hein, and R. Lyngsø. Evolving stochastic context–free grammars for RNA secondary structure prediction. *BMC Bioinformatics*, 13(1):78, 2012.

[6] R. Axelsson and M. Lange. Formal language constrained reachability and model checking propositional dynamic logics. In *International Workshop on Reachability Problems*, pages 45–57. Springer, 2011.

[7] R. Azimov and S. Grigorev. Context-free path querying by matrix multiplication. In *Proceedings of the 1st ACM SIGMOD Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA '18, pages 5:1–5:10, New York, NY, USA, 2018. ACM.

[8] R. Azimov and S. Grigorev. Path querying using conjunctive grammars. *Proceedings of the Institute for System Programming of the RAS*, 30:149–166, 01 2018.

[9] F. Bancilhon, D. Maier, Y. Sagiv, and J. D. Ullman. Magic sets and other strange ways to implement logic programs (extended abstract). In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, PODS '86, pages 1–15, New York, NY, USA, 1986. ACM.

[10] C. Barrett, K. Bisset, M. Holzer, G. Konjevod, M. Marathe, and D. Wagner. Label constrained shortest path algorithms: An experimental evaluation using transportation networks. *March*, 9:2007, 2007.

[11] C. Barrett, R. Jacob, and M. Marathe. Formal-language-constrained path problems. *SIAM Journal on Computing*, 30(3):809–837, 2000.

[12] O. Bastani, S. Anand, and A. Aiken. Specification inference using context-free language reachability. In *ACM SIGPLAN Notices*, volume 50, pages 553–566. ACM, 2015.

[13] S. Billot and B. Lang. The structure of shared forests in ambiguous parsing. In *Proceedings of the 27th Annual Meeting on Association for Computational Linguistics*, ACL '89, pages 143–151, Stroudsburg, PA, USA, 1989. Association for Computational Linguistics.

[14] D. A. Bini, M. Capovani, F. Romani, and G. Lotti. $o(n^{2.7799})$ complexity for approximate matrix multiplication. *Information Processing Letters*, 8(5):234–235, 1979.

[15] P. G. Bradford. Quickest path distances on context-free labeled graphs. In *Appear in 6-th WSEAS Conference on Computational Intelligence, Man-Machine Systems and Cybernetics*. Citeseer, 2007.

[16] P. G. Bradford. Language constrained graph problems: A microcosm of engineering research and development. In *Proceedings of the 2Nd WSEAS International Conference on Computer Engineering and Applications*, CEA'08, pages 71–76, Stevens Point, Wisconsin, USA, 2008. World Scientific and Engineering Academy and Society (WSEAS).

[17] P. G. Bradford. Efficient exact paths for dyck and semi-dyck labeled path reachability (extended abstract). In *2017 IEEE 8th Annual Ubiquitous Computing, Electronics and Mobile Communication Conference (UEMCON)*, pages 247–253, Oct 2017.

[18] P. G. Bradford and V. Choppella. Fast point-to-point dyck constrained shortest paths on a dag. In *2016 IEEE 7th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, pages 1–7. IEEE, 2016.

[19] J. A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11 (4):481—-494, 1964.

[20] T. M. Chan. All-pairs shortest paths with real weights in $o(n^3/log\, n)$ time. *Algorithmica*, 50(2):236–243, Feb 2008.

[21] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5):2075–2089, 2010.

[22] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing*, 11(3):472–492, 1982.

[23] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990.

[24] S. Dalton, N. Bell, L. Olson, and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0.

[25] T. A. Davis. Algorithm 9xx: Suitesparse:graphblas: graph algorithms in the language of sparse linear algebra. 2018.

[26] W. Dobosiewicz. A more efficient algorithm for the min-plus multiplication. *International journal of computer mathematics*, 32(1-2):49–60, 1990.

[27] W. Dyrka, M. Pyzik, F. Coste, and H. Talibart. Estimating probabilistic context-free grammars for proteins using contact map constraints. *PeerJ*, 7:e6559, Mar. 2019.

[28] N. El abbadi. An efficient storage format for large sparse matrices based on quadtree. *International Journal of Computer Applications*, 105:25–30, 11 2014.

[29] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[30] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[31] G. Gottlob, S. Ceri, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge & Data Engineering*, 1(01):146–166, jan 1989.

[32] S. Grigorev and A. Ragozina. Context-free path querying with structural representation of result. In *Proceedings of the 13th Central & Eastern European Software Engineering Conference in Russia*, CEE-SECR '17, pages 10:1–10:7, New York, NY, USA, 2017. ACM.

[33] Y. Han. Improved algorithm for all pairs shortest paths. *Information Processing Letters*, 91(5):245–250, 2004.

[34] J. Hellings. Conjunctive context-free path queries. In *Proceedings of ICDT'14*, pages 119–130, 2014.

[35] J. Hellings. Path results for context-free grammar queries on graphs. *ArXiv*, abs/1502.02242, 2015.

[36] J. Hellings. Querying for paths in graphs using context-free path queries, 2015.

[37] K. Hemerik. Towards a taxonomy for ecfg and rrpg parsing. *Proceedings of the 3rd International Conference on Language and Automata Theory and Applications*, pages 410–421, 2009.

[38] M. Holzer, M. Kutrib, and U. Leiter. Nodes connected by path languages. In G. Mauri and A. Leporati, editors, *Developments in Language Theory*, pages 276–287, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[39] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley Publishing Company, 1979.

[40] A. K. Joshi and Y. Schabes. *Tree-Adjoining Grammars*, pages 69–123. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[41] J. Kasai, B. Frank, T. McCoy, O. Rambow, and A. Nasr. TAG parsing with neural networks and vector representations of supertags. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 1712–1722, Copenhagen, Denmark, Sept. 2017. Association for Computational Linguistics.

[42] J. Kasai, R. Frank, P. Xu, W. Merrill, and O. Rambow. End-to-end graph-based TAG parsing with neural networks. In *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, pages 1181–1194, New Orleans, Louisiana, June 2018. Association for Computational Linguistics.

[43] D. Kröni and R. Schweizer. Parsing graphs: Applying parser combinators to graph traversals. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, pages 7:1–7:4, New York, NY, USA, 2013. ACM.

[44] J. Kuijpers, G. Fletcher, N. Yakovets, and T. Lindaaker. An experimental study of context-free path query evaluation methods. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management*, SSDBM '19, pages 121–132, New York, NY, USA, 2019. ACM.

[45] O. Kupferman and G. Vardi. Eulerian paths with regular constraints. In *41st International Symposium on Mathematical Foundations of Computer Science (MFCS 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[46] M. J. Kusner, B. Paige, and J. M. Hernández-Lobato. Grammar variational autoencoder. In *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ICML'17, page 1945–1954. JMLR.org, 2017.

[47] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, Jan. 2002.

[48] C. M. Medeiros, M. A. Musicante, and U. S. Costa. Ll-based query answering over rdf databases. *Journal of Computer Languages*, 51:75 – 87, 2019.

[49] M. Might and D. Darais. Yacc is dead. *CoRR*, abs/1010.5023, 2010.

[50] M. Might, D. Darais, and D. Spiewak. Parsing with derivatives: A functional pearl. *SIGPLAN Not.*, 46(9):189–195, Sept. 2011.

[51] N. Mishin, I. Sokolov, E. Spirin, V. Kutuev, E. Nemchinov, S. Gorbatyuk, and S. Grigorev. Evaluation of the context-free path querying algorithm based on matrix multiplication. In *Proceedings of the 2Nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, GRADES-NDA'19, pages 12:1–12:5, New York, NY, USA, 2019. ACM.

[52] M. Nolé and C. Sartiani. Regular path queries on massive graphs. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management*, SSDBM '16, pages 13:1–13:12, New York, NY, USA, 2016. ACM.

[53] A. Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

[54] A. Okhotin. Boolean grammars. In *Proceedings of the 7th International Conference on Developments in Language Theory*, DLT'03, pages 398–410, Berlin, Heidelberg, 2003. Springer-Verlag.

[55] A. Okhotin. On the closure properties of linear conjunctive languages. *Theor. Comput. Sci.*, 299(1-3):663–685, 2003.

[56] A. Okhotin. Conjunctive and boolean grammars: The true general case of the context-free grammars. *Computer Science Review*, 9:27–59, 8 2013.

[57] A. Okhotin. Parsing by matrix multiplication generalized to boolean grammars. *Theor. Comput. Sci.*, 516:101–120, Jan. 2014.

[58] A. S. Okhotin. Conjunctive grammars and systems of language equations. *Programming and Computer Software*, 28(5):243–249, Sep 2002.

[59] V. Y. Pan. Strassen's algorithm is not optimal trilinear technique of aggregating, uniting and canceling for constructing fast algorithms for matrix operations. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 166–176. IEEE, 1978.

[60] P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via cfl reachability. In *SAS*, volume 6, pages 88–106. Springer, 2006.

[61] J. Rehof and M. Fähndrich. Type-base flow analysis: From polymorphic subtyping to cfl-reachability. *SIGPLAN Not.*, 36(3):54–66, Jan. 2001.

[62] J. G. Rekers. *Parser generation for interactive environments*. PhD thesis, Universiteit van Amsterdam, 1992.

[63] T. Reps. Program analysis via graph reachability. In *Proceedings of the 1997 International Symposium on Logic Programming*, ILPS '97, pages 5–19, Cambridge, MA, USA, 1997. MIT Press.

[64] T. Reps, S. Horwitz, M. Sagiv, and G. Rosay. Speeding up slicing. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '94, pages 11–20, New York, NY, USA, 1994. Association for Computing Machinery.

[65] B. Roy. Transitivité et connexité. *Comptes Rendus Hebdomadaires Des Seances De L Academie Des Sciences*, 249(2):216–218, 1959.

[66] A. Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.

[67] E. Scott and A. Johnstone. Right nulled glr parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, July 2006.

[68] E. Scott and A. Johnstone. Gll parsing. *Electron. Notes Theor. Comput. Sci.*, 253(7):177–189, Sept. 2010.

[69] E. Scott and A. Johnstone. Multiple lexicalisation (a java based study). In *Proceedings of the 12th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2019, page 71–82, New York, NY, USA, 2019. Association for Computing Machinery.

[70] E. Scott, A. Johnstone, and R. Economopoulos. Brnglr: A cubic tomita-style glr parsing algorithm. *Acta Inf.*, 44(6):427–461, Sept. 2007.

[71] H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191 – 229, 1991.

[72] P. Sevon and L. Eronen. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics*, 5, 06 2008.

[73] E. Shemetova and S. Grigorev. Path querying on acyclic graphs using boolean grammars. *Proceedings of the Institute for System Programming of RAS*, 31(4):211–226, 2019.

[74] V. Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[75] Sylvain Salvati. Multiple context-free grammars.

[76] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43(4):195–199, 1992.

[77] T. Takaoka. A faster algorithm for the all-pairs shortest path problem and its application. In *Computing and Combinatorics*, volume 3106, pages 278–289, 2004.

[78] T. Takaoka. An $o(n^3 \log\log n / \log n)$ time algorithm for the all-pairs shortest path problem. *Information Processing Letters*, 96:155–161, 2005.

[79] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13:31–46, 1987.

[80] M. Tomita. Graph-structured stack and natural language parsing. In *26th Annual Meeting of the Association for Computational Linguistics*, pages 249–257, Buffalo, New York, USA, June 1988. Association for Computational Linguistics.

[81] L. G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, Apr. 1975.

[82] E. Verbitskaia, S. Grigorev, and D. Avdyukhin. Relaxed parsing of regular approximations of string-embedded languages. In M. Mazzara and A. Voronkov, editors, *Perspectives of System Informatics*, pages 291–302, Cham, 2016. Springer International Publishing.

[83] E. Verbitskaia, I. Kirillov, I. Nozkin, and S. Grigorev. Parser combinators for context-free path querying. In *Proceedings of the 9th ACM SIGPLAN International Symposium on Scala*, Scala 2018, pages 13–23, New York, NY, USA, 2018. ACM.

[84] C. B. Ward and N. M. Wiegand. Complexity results on labeled shortest path problems from wireless routing metrics. *Comput. Netw.*, 54(2):208–217, Feb. 2010.

[85] C. B. Ward, N. M. Wiegand, and P. G. Bradford. A distributed context-free language constrained shortest path algorithm. In *2008 37th International Conference on Parallel Processing*, pages 373–380. IEEE, 2008.

[86] S. Warshall. A theorem on boolean matrices. 1962.

[87] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[88] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 155–165, New York, NY, USA, 2011. Association for Computing Machinery.

[89] M. Yannakakis. Graph-theoretic methods in database theory. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 230–242. ACM, 1990.

[90] Q. Zhang, X. Xiao, C. Zhang, H. Yuan, and Z. Su. Efficient subcubic alias analysis for c. *SIGPLAN Not.*, 49(10):829–845, Oct. 2014.

[91] X. Zhang, Z. Feng, X. Wang, G. Rao, and W. Wu. Context-free path queries on rdf graphs. In P. Groth, E. Simperl, A. Gray, M. Sabou, M. Krötzsch, F. Lecue, F. Flöck, and Y. Gil, editors, *The Semantic Web – ISWC 2016*, pages 632–648, Cham, 2016. Springer International Publishing.

[92] X. Zheng and R. Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM.

[93] R. Zier-Vogel and M. Domaratzki. Rna pseudoknot prediction through stochastic conjunctive grammars. *Computability in Europe 2013. Informal Proceedings*, pages 80–89, 2013.

[94] U. Zwick. A slightly improved sub-cubic algorithm for the all pairs shortest paths problem with real edge lengths. In *International Symposium on Algorithms and Computation*, pages 921–932. Springer, 2004.