О достижимости с ограничениями в терминах формальных языков

Семён Григорьев

6 января 2023 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.

Например, в машинном обучении использование формальных грамматик позволяет передать искусственной генеративной нейронной сети, предназначенной для построения цепочек с определёнными свойствами, знания о синтаксической структуре этих цепочек, что позволяет существенно упростить процесс обучения и повысить качество результата [45]. Вместе с этим, развиваются подходы, позволяющие нейронным сетям наоборот извлекать синтаксическую структуру (строить дерево вывода) для входных цепочек [41, 42].

В биоинформатике формальные грамматики нашли широкое применение для описания особенностей вторичной структуры геномных и белковых последовательностей [27, 4, 84]. 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 анализа 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 графовых баз данных и для статического анализа кода.

Одна из классических задач, связанных с анализом графов — это поиск путей в графе. Возможны различные формулировки этой задачи. В некоторых случаях 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, в качестве ограничений можно 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. Базовое 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 конкатенацией меток их рёбер, принадлежат заданному языку. Такой класс задач будем называть задачами поиска путей с ограничениям в терминах формальных языков.

Подобный класс задач часто возникает в областях, связанных с анализом граф-структурированных данных и активно исследуется [11, 5, 38, 74, 10, 11]. Исследуются как классы языков, применяемых для задания ограничений, так и различные постановки задачи.

Граф-структурированные 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 данных и так далее. Оказывается, что поиск путей в специального вида графах с использованием ограничений в терминах формальных языков позволяет исследовать некоторые нетривиальные свойства программы. Например проводить межпроцедурный анализ указателей или анализ алиасов [83, 78, 81], строить срезы программ [56], проводить анализ типов [53].

В данной работе представлен ряд алгоритмов для поиска путей с ограничениями в терминах формальных языков. Основной акцент будет сделан на контекстно-свободных языках, однако будут затронуты и другие классы: регулярные, многокомпонентные контекстно-свободные (Multiple Context-Free Languages, MCFL [63]) и конъюнктивные языки. Будет показано, что теория формальных языков и алгоритмы синтаксического 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. В начале (в части 2) мы рассмотрим основные понятия из теории графов, необходимые в данной работе. Данные разделы являются подготовительными и не обязательны к прочтению, если такие понятия как *ориентированный граф* и *матрица смежности* уже известны читателю. Более того, они лишь вводят определения, подразумевая,

что более детальное изучение соответствующих разделов науки остается за рамками этой работы и скорее всего уже проделано читателем. Затем, в главе 3 мы введём основные понятия из теории формальных языков. Далее, в главе 7 рассмотрим различные варианты постановки задачи поиска путей с ограничениями в терминах формальных языков, обсудим базовые свойства задач, её разрешимость в различных постановках и т.д.. И в итоге зафиксируем постановку, которую будем изучать далее. После этого, в главах 9–14 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.

Глава 1

Некоторые понятия линейной алгебры[1]

При изложении ряда алгоритмов будут активно использоваться некоторые понятия и инструменты линейной алгебры, такие как моноид, полукольцо или матрица. В данном разделе необходимые понятия будут определены и приведены некоторые примеры соответствующих конструкций. Для более глубокого изучения материала рекомендуются обратиться к соответствующим разделам алгебры.

1.1 Бинарные операции и их свойства

Введём понятие *бинарной операции* и рассмотрим некоторые её свойства, такие как *коммутативность* и *ассоциативность*.

Определение 1.1.1. *Функцией* будем называть бинарное отношение на двух множествах X и Y, такое, что каждому элементу из X сопоставляется ровно один элемент из Y. Запись $f : X \to Y$ как раз и обозначает, что функция f сопоставляет элементы из X элементам из Y.

Определение 1.1.2. Для функции $f : X \to Y$, множество X называется *областью определения функции* или *доменом функции*.

[1]Неообходимо понимать, что, с одной строны, в данном разделе рассматриваются самые базовые понятия, которые даются практически в любом учебнике алгебры. С другой же стороны, определения данных понятий оказываются весьма вариативными и часто вызывают дискуссии. Напрмиер, интересный анализ тонкостей определения группы можно найти в первом и втором параграфах первого раздела книги Николая Александровича Вавилова "Конкретная теория групп" [87]. Мы же дадим определения, удобные для дальнейшего изложения материала.

Определение 1.1.3. Для функции $f : X \to Y$, множество Y называется *областью значений функции* или *кодоменом функции*.

Определение 1.1.4. Функцию, принимающую два аргумента, $f : S \times K \to Q$ будем называть *двухместной* или *функцией арности два*. Для записи таких функций будем использовать типичную нотацию: $c = f(a, b)$.

Определение 1.1.5. *Бинарная операция* — это двухместная функция, от которой дополнительно требуется, чтобы оба аргумента и результат лежали в одном и том же множестве: $f : S \times S \to S$. В таком случае говорят, что бинарная операция определена на некотором множестве S. Для обозначения произвольной бинарной операции будем использовать символ $\circ$ и пользоваться инфиксной нотацией для записи: $c = a \circ b$.

Определение 1.1.6. *Внешняя бинарная операция* — это бинарная операция, у которой аргументы лежат в разных множествах, при этом результат — в одном из этих множеств. Иными словами $\circ : K \times S \to S$, где K может не совпадать с S — внешняя бинарная операция.

Необходимо помнить, что как функции, так и бинарные операции, могут быть частично определёнными (частичные функции, частичные бинарные операции). Типичным примером частично определённой бинарной операции является деление на целых числах: она не определена, если второй аргумент равен нулю.

Бинарные операции могут обладать некоторыми дополнительными свойствами, такими как *коммутативность* или *ассоциативность*, позволяющими преобразовывать выражения, составленные с использованием этих операций.

Определение 1.1.7. Бинарная операция $\circ : S \times S \to S$ называется *коммутативной*, если для любых $x_1 \in S, x_2 \in S$ верно, что $x_1 \circ x_2 = x_2 \circ x_1$.

Пример 1.1.1. Рассмотрим несколько примеров коммутативных и некоммутативных операций.

- Операция сложения на целых числах $+$ является коммутативной: известный ещё со школы перестановочный закон сложения.
- Операция конкатенации на строках $\cdot$ не является коммутативной:

$$\text{“}ab\text{”} \cdot \text{“}c\text{”} = \text{“}abc\text{”} \neq \text{“}cab\text{”} = \text{“}c\text{”} \cdot \text{“}ab\text{”}.$$

- Операция умножения на целых числах является коммутативной: известный ещё со школы перестановочный закон умножения.

- Операция умножения матриц (над целыми числами) $\cdot$ не является коммутативной:

$$\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix} \neq \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}.$$

Определение 1.1.8. Бинарная операция $\circ : S \times S \to S$ называется *ассоциативной*, если для любых $x_1 \in S, x_2 \in S, x_3 \in S$ верно, что $(x_1 \circ x_2) \circ x_3 = x_1 \circ (x_2 \circ x_3)$. Иными словами, для ассоциативной операции результат вычислений не зависит от порядка применения операций.

Пример 1.1.2. Рассмотрим несколько примеров ассоциативных и неассоциативных операций.

- Операция сложения на целых числах $+$ является ассоциативной.
- Операция умножения на целых числах является ассоциативной.
- Операция конкатенации на строках $\cdot$ является ассоциативной:

$$(\text{“}a\text{”} \cdot \text{“}b\text{”}) \cdot \text{“}c\text{”} = \text{“}a\text{”} \cdot (\text{“}b\text{”} \cdot \text{“}c\text{”}) = \text{“}abc\text{”}.$$

- Операция возведения в степень (над целыми числами) ^ не является ассоциативной:

$$(2\text{^}2)\text{^}3 = 4\text{^}3 = 64 \neq 256 = 2\text{^}8 = 2\text{^}(2\text{^}3).$$

Определение 1.1.9. Говорят, что бинарная операция $\otimes : S \times S \to S$ является *дистрибутивной* относительно бинарной операции $\oplus : S \times S \to S$, если

1. Для любых $x_1, x_2, x_3 \in S, x_1 \otimes (x_2 \oplus x_3) = (x_1 \otimes x_2) \oplus (x_1 \otimes x_3)$ (дистрибутивность слева).
2. Для любых $x_1, x_2, x_3 \in S, (x_2 \oplus x_3) \otimes x_1 = (x_2 \otimes x_1) \oplus (x_3 \otimes x_1)$ (дистрибутивность справа).

Если операция $\otimes$ является коммутативной, то дистрибутивность слева и справа равносильны.

Пример 1.1.3. Рассмотрим несколько примеров дистрибутивных операций.

- Умножение целых чисел дистрибутивно относительно сложения и вычитания: классический *распределительный закон*, знакомый всем со школы.

- Операция деления (допустим, на действительных числах) не коммутативна. При этом, она дистрибутивна справа относительно сложения и вычитания, но не дистрибутивна слева.

$$(a + b)/c = (a/c) + (b/c)$$

но

$$c/(a + b) \neq (c/a) + (c/b)^2.$$

Определение 1.1.10. Бинарная операция $\circ : S \times S \to S$ называется *идемпотентной*, если для любого $x \in S$ верно, что $x \circ x = x$.

Пример 1.1.4. Рассмотрим несколько примеров идемпотентных операций.

- Операция объединения множеств $\cup$ является идемпотентной: для любого множества S верно, что $S \cup S = S$.
- Операция сложения на целых числах не является идемпотентной.
- Операции "логическое и" $\wedge$ и "логическое или" $\vee$ являются идемпотентными.
- Операция "исключающее или" не является идемпотентной.

Определение 1.1.11. Пусть есть коммутативная бинарная операция $\circ$ на множестве S. Говорят, что $x \in S$ является *нейтральным элементом* по операции $\circ$, если для любого $y \in S$ верно, что $x \circ y = y \circ x = y$. Если бинарная операция не является коммутативной, то можно определить *нейтральный слева* и *нейтральный справа* элементы по аналогии.

1.2 Полугруппа

Определение 1.2.1. Множество S с заданной на нём ассоциативной бинарной операцией $\cdot : S \times S \to S$ называется *полугруппой* и обозначается $(S, \cdot)$. Если операция $\cdot$ является коммутативной, то говорят о *коммутативной полугруппе*.

Пример 1.2.1. Приведём несколько примеров полугрупп.

[2]Здесь может быть уместно вспомнить правила сложения дробей. Дроби с общим знаминателем складывать проще как раз из-за дистрибутивности справа.

- Множество положительных целых чисел с операцией сложения является коммутативной полугруппой.

- Множество целых чисел с операцией взятия наибольшего из двух (max) является коммутативной полугруппой.

- Множество всех строк конечной длины без пустой строки[3] над фиксированным алфавитом Σ с операцией конкатенации является полугруппой. Так как конкатенация на строках не является коммутативной операцией, то и полугруппа не является коммутативной.

1.3 Моноид

Определение 1.3.1. *Моноидом* называется полугруппа с нейтральным элементом. Если операция является коммутативной, то можно говорить о *коммутативном моноиде*.

Пример 1.3.1. Приведём примеры моноидов, построенных на основе полугрупп из предыдущего раздела.

- Неотрицательные целые числа (или же натуральные числа с нулём) с операцией сложения являются моноидом. Нейтральный элемент — 0.

- Целые числа, дополненные значением $-\infty$ ("минус-бесконечность") с операцией взятия наибольшего из двух (max) являются моноидом. Нейтральный элемент — $-\infty$.

- Множество всех строк конечной длины с пустой строкой (строка длины 0) над фиксированным алфавитом Σ и операцией конкатенации является моноидом. Нейтральный элемент — пустая строка.

- Квадратные неотрицательные матрицы[4] фиксированного размера с операцией умножения задают моноид. Нейтральный элемент — единичная матрица.

[3] Множество всех строк конечной длины с пустой строкой также является полугруппой. Однако, такая структура является ещё и моноидом, что будет показано далее.

[4] Неотрицательной называется матрица, все элементы которой не меньше нуля.

1.4 Группа

Определение 1.4.1. Непустое[5] множество G с заданной на нём бинарной операцией $\circ : G \times G \to G$ называется *группой* $(G, \circ)$, если выполнены следующие аксиомы:

1. ассоциативность: $\forall (a, b, c \in G) \colon (a \circ b) \circ c = a \circ (b \circ c)$;
2. наличие нейтрального элемента: $\exists e \in G \quad \forall a \in G \colon (e \circ a = a \circ e = a)$;
3. наличие обратного элемента: $\forall a \in G \quad \exists a^{-1} \in G \colon (a \circ a^{-1} = a^{-1} \circ a = e)$.

Иными словами, группа — это моноид с дополнительным требованием наличия обратных элементов.

Определение 1.4.2. Если операция $\circ$ коммутативна, то говорят, что группа *Абелева*.

Пример 1.4.1. Рассмотрим несколько примеров групп.

- Целые числа $\mathbb{Z}$ с операцией сложения $+$ являются группой. Получается дополнением моноида из предыдущего раздела обратными по сложению элементами.
- Целые числа $\mathbb{Z}$ без нуля[6] с операцией умножения $\cdot$ не являются группой, так как нет обратных по умножению. Действительно, возьмём $a = 3$, тогда должен существовать $a^{-1} \in \mathbb{Z}$, такой что $3 \cdot a^{-1} = 1$. Видим, что $a^{-1} = \frac{1}{3}$, но $\frac{1}{3} \notin \mathbb{Z}$.
- Множество обратимых[7] матриц с операцией матричного умножения задают группу.

[5] Требование непустоты здесь, как и далее, в определениях полукольца и кольца — дискуссионный вопрос.

[6] При наличии нуля возникают трудности с нейтральным элементом. Логично считать 1 — нейтральным по умножению, однако $0 \cdot 1 = 0$, а не 1, как того требует определение.

[7] Квадратная матрица M называется обратимой, если существует матрица N, называемая обратной, такая что $M \cdot N = N \cdot MI$, где I — единичная матрица. К сожалению, не все матрицы являются обратимыми, потому, чтобы сконструировать группу, нам приходится требовать обратимость явно.

1.5 Полукольцо

Определение 1.5.1. Непустое множество R с двумя бинарными операциями $\oplus\colon R \times R \to R$ (часто называют сложением) и $\otimes\colon R \times R \to R$ (часто называют умножением) называется *полукольцом*, если выполнены следующие условия.

1. $(R, \oplus)$ — это коммутативный моноид, нейтральный элемент которого — $\mathbb{0}$. Для любых $a, b, c \in R$:
   - $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
   - $\mathbb{0} \oplus a = a \oplus \mathbb{0} = a$
   - $a \oplus b = b \oplus a$
2. $(R, \otimes)$ — это моноид, нейтральный элемент которого — $\mathbb{1}$. Для любых $a, b, c \in R$:
   - $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
   - $\mathbb{1} \otimes a = a \otimes \mathbb{1} = a$
3. $\otimes$ дистрибутивно слева и справа относительно $\oplus$:
   - $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$
   - $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$
4. $\mathbb{0}$ является *аннигилятором* по умножению:
   - $\forall a \in R : \mathbb{0} \otimes a = a \otimes \mathbb{0} = \mathbb{0}$

Если операция $\otimes$ коммутативна, то говорят о коммутативном полукольце. Если операция $\oplus$ идемпотентна, то говорят об идемпотентном полукольце.

Пример 1.5.1. Рассмотрим пример полукольца, а заодно покажем, что левая и правая дистрибутивность могут существовать независимо для некоммутативного умножения[8].

В качестве R возьмём множество множеств строк конечной длины над некоторым алфавитом Σ. В качестве сложения возьмём теоретико-множественное

[8]Хороший пример того, почему левую и правую дистрибутивность в случае некоммутативного умножения нужно проверять независимо (правда, для колец), приведён Николаем Александровичем Вавиловым в книге “Конкретная теория колец” на странице 6 [88].

объединение: $\oplus \equiv \cup$. Нейтральный элемент по сложению — это пустое множество ($\varnothing$). В качестве умножения возьмём конкатенацию множеств ($\otimes \equiv \odot$) и определим её следующим образом:

$$S_1 \odot S_2 = \{w_1 \cdot w_2 \mid w_1 \in S_1, w_2 \in S_2\}$$

, где $\cdot$ — конкатенация строк. Нейтральным элементом по умножению будет являться множество из пустой строки: $\{\varepsilon\}$, где ε — обозначение для пустой строки.

Проверим, что $(R, \cup, \odot)$ действительно полукольцо по нашему определению.

1. $(R, \cup)$ — действительно коммутативный моноид с нейтральным элементом $\varnothing$. Для любых $a, b, c \in R$ по свойствам теоретико-множественного объединения верно:

   - $(a \cup b) \cup c = a \cup (b \cup c)$
   - $\varnothing \cup a = a \cup \varnothing = a$
   - $a \cup b = b \cup a$.

2. $(R, \odot)$ — действительно моноид с нейтральным элементом $\{\varepsilon\}$. Для любых $a, b, c \in R$:

   - $(a \odot b) \odot c = a \odot (b \odot c)$ по определению $\odot$
   - $\{\varepsilon\} \odot a = \{\varepsilon \cdot w \mid w \in a\} = \{w \mid w \in a\} = a \odot \{\varepsilon\} = a$

   Вообще говоря, сконструированный нами моноид не является коммутативным: легко проверить, например, что существуют непустые $a, b \in R, a \neq b, a \neq \{\varepsilon\}, b \neq \{\varepsilon\}$: $a \cdot b \neq b \cdot a$ по причине некоммутативности конкатенации строк.

3. $\odot$ дистрибутивно слева и справа относительно $\cup$:

   - $a \odot (b \cup c) = \{w_1 \cdot w_2 \mid w_1 \in a, w_2 \in b \cup c\} = \{w_1 \cdot w_2 \mid w_1 \in a, w_2 \in b\} \cup \{w_1 \cdot w_2 \mid w_1 \in a, w_2 \in c\} = (a \odot b) \cup (a \odot c)$
   - Аналогично, $(a \cup b) \odot c = (a \odot c) \cup (b \odot c)$

   При этом, в общем случае, $a \odot (b \cup c) \neq (b \cup c) \odot a$ из-за некоммутативности операции $\odot$. Действительно,

$$\{\text{“}a\text{”}\} \odot (\{\text{“}b\text{”}\} \cup \{\text{“}c\text{”}\}) = \{\text{“}a\text{”}\} \odot \{\text{“}b\text{”}\ , \text{“}c\text{”}\} = \{\text{“}ab\text{”}\ , \text{“}ac\text{”}\} \neq$$
$$\neq (\{\text{“}b\text{”}\} \cup \{\text{“}c\text{”}\}) \odot \{\text{“}a\text{”}\} = \{\text{“}b\ \text{“}c\text{”}\} \odot \{\text{“}a\text{”}\} = \{\text{“}ba\text{”}\ , \text{“}ca\text{”}\}$$

4. $\varnothing$ является *аннигилятором* по умножению: для любого $a \in R$ верно, что $\varnothing \odot a = \{w_1 \cdot w_2 \mid w_1 \in \varnothing, w_2 \in a\} = \{w_1 \cdot w_2 \mid w_1 \in a, w_2 \in \varnothing\} = a \odot \varnothing = \varnothing$

1.6 Кольцо

Определение 1.6.1. Непустое множество R с двумя бинарными операциями $\oplus\colon R \times R \to R$ (умножение) и $\otimes\colon R \times R \to R$ (сложение) называется *кольцом*, если выполнены следующие условия.

1. $(R, \oplus)$ — это Абелева группа, нейтральный элемент которой — $\mathbb{0}$. Для любых $a, b, c \in R$:

   - $(a \oplus b) \oplus c = a \oplus (b \oplus c)$
   - $\mathbb{0} \oplus a = a \oplus \mathbb{0} = a$
   - $a \oplus b = b \oplus a$
   - для любого $a \in R$ существует $-a \in R$, такое что $a + (-a) = \mathbb{0}$.

   В последнем пункте кроется отличие от полукольца.

2. $(R, \otimes)$ — это моноид, нейтральный элемент которого — $\mathbb{1}$. Для любых $a, b, c \in R$:

   - $(a \otimes b) \otimes c = a \otimes (b \otimes c)$
   - $\mathbb{1} \otimes a = a \otimes \mathbb{1} = a$

3. $\otimes$ дистрибутивно слева и справа относительно $\oplus$:

   - $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$
   - $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$

Заметим, что мультипликативное свойство $\mathbb{0}$ (быть аннигилятором по умножению) не указыватеся явно, так как может быть выведено из остальных утверждений. Действительно,

1. $a \otimes \mathbb{0} = a \otimes (\mathbb{0} \oplus \mathbb{0})$, так как $\mathbb{0}$ — нейтральный по сложению, то $\mathbb{0} \oplus \mathbb{0} = \mathbb{0}$

2. Воспользуемся дистрибутивностью: $a \otimes (\mathbb{0} \oplus \mathbb{0}) = a \otimes \mathbb{0} \oplus a \otimes \mathbb{0}$. В итоге: $a \otimes \mathbb{0} = a \otimes \mathbb{0} \oplus a \otimes \mathbb{0}$

3. Так как у нас есть группа по сложению, то для любого a существует обратный элемент $a^{-1}, a \oplus a^{-1} = \mathbb{0}$. Прибавим $a^{-1} \otimes \mathbb{0}$ к левой и правой части равенства[9], полученного на предыдущем шаге:

$$a \otimes \mathbb{0} \oplus a^{-1} \otimes \mathbb{0} = a \otimes \mathbb{0} \oplus a \otimes \mathbb{0} \oplus a^{-1} \otimes \mathbb{0}.$$

4. Воспользуемся дистрибутивностью и ассоциативностью.

$$\begin{gathered}(a \oplus a^{-1}) \otimes \mathbb{0} = a \otimes \mathbb{0} \oplus (a \oplus a^{-1}) \otimes \mathbb{0} \\ \mathbb{0} \otimes \mathbb{0} = a \otimes \mathbb{0} \oplus \mathbb{0} \otimes \mathbb{0} \\ \mathbb{0} = a \otimes \mathbb{0}\end{gathered}$$

5. Аналогично можно доказать, что $\mathbb{0} = \mathbb{0} \otimes a$.

1.7 Матрицы и вектора

К определению матрицы мы подойдём структурно, так как в дальнейшем будем сопоставлять эту структуру с объектами различной природы, а значит определение матрицы через какой-либо из этих объектов (например через квадратичные формы) будет менее удобным.

Договоримся, что *алгебраическая структура* — это собирательное название для объектов вида "множество с набором операций" (например, кольцо, моноид, группа и т.д.), а соответствующее множество будем назвать *носителем* этой структуры.

Определение 1.7.1. Предположим, что у нас есть некоторая алгебраическая структура с носителем S. Тогда *матрицей* будем называть прямоугольный массив размера $n \times m, n > 0, m > 0$, заполненный элементами из S.

Говорят, что n — это высота матрицы или количество строк в ней, а m — ширина матрицы или количество столбцов.

При доступе к элементам матрицы используются их индексы. При этом нумерация ведётся с левого верхнего угла, первым указывается строка, вторым — столбец. В нашей работе мы будем использовать "программистскую" традицию и нумеровать строки и столбцы с нуля[10].

[9] Обычно данное действие воспринимается как очевидное, но, строго говоря, оно требует аккуратного введения структур с равенством и соответствующих аксиом.

[10] В противоположность "математической" традиции нумеровать строки и столбцы с единицы. Стоит, правда, отметить, что в некоторых языках программирования (например, Fortran или COBOL) жива "математическая" традиция.

Пример 1.7.1 (Матрица)**.** Пусть есть моноид $(S, \cdot)$, где S — множество строк конечной длины над алфавитом {a,b,c}. Тогда можно построить, например, следующую матрицу 2×3.

$$M_{2\times 3} = \begin{pmatrix} \text{“}a\text{”} & \text{“}ba\text{”} & \text{“}cb\text{”} \\ \text{“}ac\text{”} & \text{“}bab\text{”} & \text{“}b\text{”} \end{pmatrix}$$

$$M_{2\times 3}[1,1] = \text{“}bab\text{”}$$

К определению вектора мы также подойдём структурно.

Определение 1.7.2. *Вектором* будем называть матрицу, хотя бы один из размеров которой равен единице. Если единице равна высота матрицы, то это *вектор-строка*, если же единице равна ширина матрицы, то это *вектор-столбец*.

Операции над матрицами можно условно разделить на две группы:

- *структурные* — не зависящие от алгебраической структуры, над которой строилась матрица, и работающие только с её структурой;
- *алгебраические* — определение таковых опирается на свойства алгебраической структуры, над которой построена матрица.

Примерами структурных операций является *транспонирование*, *взятие подматрицы* и *взятие элемента по индексу*.

Определение 1.7.3 (Транспонирование матрицы)**.** Пусть дана матрица $M_{n\times m}$. Тогда результат её *транспонирования*, это такая матрица $M'_{m\times n}$, что $M'[i,j] = M[j,i]$ для всех $0 \le i \le m-1$ и $0 \le j \le n-1$.

Операцию транспонирования принято обозначать как M^T.

Пример 1.7.2.

$$\begin{pmatrix} \text{“}a\text{”} & \text{“}ba\text{”} & \text{“}cb\text{”} \\ \text{“}ac\text{”} & \text{“}bab\text{”} & \text{“}b\text{”} \end{pmatrix}^T = \begin{pmatrix} \text{“}a\text{”} & \text{“}ac\text{”} \\ \text{“}ba\text{”} & \text{“}bab\text{”} \\ \text{“}cd\text{”} & \text{“}b\text{”} \end{pmatrix}$$

Определение 1.7.4 (Прямая сумма матриц)**.** Пусть даны матрицы $M_{n_1\times m_1}$ и $N_{n_2\times m_2}$. Тогда *прямой суммой* этих матриц называется матрица $L_{n_1+n_2\times m_1+m_2}$ вида

$$L = \begin{bmatrix} M & 0 \\ 0 & N \end{bmatrix}$$

Где 0 обозначает нулевой блок. Прямая сумма обозначается $L = M \bigoplus N$.

Определение 1.7.5 (Взятие подматрицы)**.** Пусть дана матрица $M_{n\times m}$. Тогда $M_{n\times m}[i_0..i_1, j_0..j_1]$ — это такая $M'_{i_1-i_0+1,j_1-j_0+1}$ что $M'[i,j] = M[i_0+i, j_0+j]$ для всех $0 \le i \le i_1 - i_0 + 1$ и $0 \le j \le j_1 - j_0 + 1$.

Пример 1.7.3.

$$\begin{pmatrix} \text{“}a\text{”} & \text{“}ba\text{”} & \text{“}cb\text{”} \\ \text{“}ac\text{”} & \text{“}bab\text{”} & \text{“}b\text{”} \end{pmatrix}[0..1, 1..2] = \begin{pmatrix} \text{“}ba\text{”} & \text{“}cb\text{”} \\ \text{“}bab\text{”} & \text{“}b\text{”} \end{pmatrix}$$

Определение 1.7.6. *Взятие элемента по индексу* — это частный случай взятия подматрицы, когда начало и конец “среза” совпадают: $M[i,j] = M[i..i, j..j]$

Пример 1.7.4.

$$\begin{pmatrix} \text{“}a\text{”} & \text{“}ba\text{”} & \text{“}cb\text{”} \\ \text{“}ac\text{”} & \text{“}bab\text{”} & \text{“}b\text{”} \end{pmatrix}[0,1] = \text{“}ba\text{”}$$

Из алгебраических операций над матрицами нас в дальнейшем будут интересовать *поэлементные операции*, *скалярные операции*, *матричное умножение*, *произведение Кронекера*.

Определение 1.7.7 (Поэлементные операции)**.** Пусть $G = (S, \circ)$ — полугруппа[11], $M_{n\times m}, N_{n\times m}$ — две матрицы одинакового размера над этой полугруппой. Тогда $ewise(M, N, \circ) = P_{n\times m}$,такая, что $P[i,j] = M[i,j] \circ N[i,j]$.

Пример 1.7.5. Пусть G — полугруппа строк с конкатенацией $\cdot$,

$$M = \begin{pmatrix} \text{“}a\text{”} & \text{“}ba\text{”} & \text{“}cb\text{”} \\ \text{“}ac\text{”} & \text{“}bab\text{”} & \text{“}b\text{”} \end{pmatrix},$$

$$N = \begin{pmatrix} \text{“}c\text{”} & \text{“}aa\text{”} & \text{“}b\text{”} \\ \text{“}a\text{”} & \text{“}bac\text{”} & \text{“}bb\text{”} \end{pmatrix}.$$

Тогда

$$ewise(M, N, \cdot) = \begin{pmatrix} \text{“}ac\text{”} & \text{“}baaa\text{”} & \text{“}cbb\text{”} \\ \text{“}aca\text{”} & \text{“}babbac\text{”} & \text{“}bbb\text{”} \end{pmatrix}.$$

Определение 1.7.8 (Скалярная операция)**.** Пусть $G = (S, \circ)$ — полугруппа, $M_{n\times m}$ — матрица над этой полугруппой, $x \in S$. Тогда $scalar(M, x, \circ) = P_{n\times m}$, такая, что $P[i,j] = M[i,j] \circ x$, а $scalar(x, M, \circ) = P_{n\times m}$, такая, что $P[i,j] = x \circ M[i,j]$.

[11]Здесь, как и в дальнейшем, требование к структуре быть полугруппой не обязательно. Оно лишь позволяет нам получить ассоциативность соответствующих операций над матрицами, что может оказаться полезным при дальнейшей работе.

Пример 1.7.6. Пусть G — полугруппа строк с конкатенацией $\cdot$, $x =$ "c",

$$M = \begin{pmatrix} \text{“}a\text{”} & \text{“}ba\text{”} & \text{“}cb\text{”} \\ \text{“}ac\text{”} & \text{“}bab\text{”} & \text{“}b\text{”} \end{pmatrix}.$$

Тогда

$$scalar(M, x, \cdot) = \begin{pmatrix} \text{“}ac\text{”} & \text{“}bac\text{”} & \text{“}cbc\text{”} \\ \text{“}acc\text{”} & \text{“}babc\text{”} & \text{“}bc\text{”} \end{pmatrix},$$

$$scalar(x, M, \cdot) = \begin{pmatrix} \text{“}ca\text{”} & \text{“}cba\text{”} & \text{“}ccb\text{”} \\ \text{“}cac\text{”} & \text{“}cbab\text{”} & \text{“}cb\text{”} \end{pmatrix}.$$

Определение 1.7.9 (Матричное умножение)**.** Пусть $G = (S, \oplus, \otimes)$ — полукольцо, $M_{n\times m}, N_{m\times k}$ — две матрицы над этим полукольцом. Тогда $M \cdot N = P_{n\times k}$, такая, что $P[i,j] = \bigoplus_{0\le l<m} M[i,l] \otimes N[l,j]$.

Пример 1.7.7. Пусть G — полукольцо из примера 1.5.1,

$$M = \begin{pmatrix} \{\text{“}a\text{”}\} & \{\text{“}a\text{”}\} \\ \{\text{“}b\text{”}\} & \{\text{“}b\text{”}\} \end{pmatrix},$$

$$N = \begin{pmatrix} \{\text{“}c\text{”}\} \\ \{\text{“}d\text{”}\} \end{pmatrix}.$$

Тогда

$$M \cdot N = \begin{pmatrix} \{\text{“}a\text{”} \cdot \text{“}c\text{”}\} \cup \{\text{“}a\text{”} \cdot \text{“}d\text{”}\} \\ \{\text{“}b\text{”} \cdot \text{“}c\text{”}\} \cup \{\text{“}b\text{”} \cdot \text{“}d\text{”}\} \end{pmatrix} = \begin{pmatrix} \{\text{“}ac\text{”} \text{ , “}ad\text{”}\} \\ \{\text{“}bc\text{”} \text{ , “}bd\text{”}\} \end{pmatrix}.$$

Определение 1.7.10 (Произведение Кронекера)**.** Пусть $G = (S, \circ)$ — полугруппа, $M_{m\times n}$ и $N_{p\times q}$ — две матрицы над этой полугруппой. Тогда произведение Кронекера или тензорное произведение матриц M и N — это блочная матрица C размера $mp \times nq$, вычисляемая следующим образом:

$$C = A \otimes B = \begin{pmatrix} scalar(M[0,0], N, \circ) & \cdots & scalar(M[0,n-1], N, \circ) \\ \vdots & \ddots & \vdots \\ scalar(M[m-1,0], N, \circ) & \cdots & scalar(M[m-1,n-1], N, \circ) \end{pmatrix}$$

Утверждение 1.7.1. Произведение Кронекера не является коммутативным. При этом всегда существуют две матрицы перестановок P и Q такие, что $A \otimes B = P(B \otimes A)Q$.

Пример 1.7.8. Возьмём в качестве полугруппы целые числа с умножением.

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$N = \begin{pmatrix} 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

Тогда

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

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

$$= \left(\begin{array}{cccc|cccc} 5 & 6 & 7 & 8 & 10 & 12 & 14 & 16 \\ 9 & 10 & 11 & 12 & 18 & 20 & 22 & 24 \\ 13 & 14 & 15 & 16 & 26 & 28 & 30 & 32 \\ \hline 15 & 18 & 21 & 24 & 20 & 24 & 28 & 32 \\ 27 & 30 & 33 & 36 & 36 & 40 & 44 & 48 \\ 39 & 42 & 45 & 48 & 52 & 56 & 60 & 64 \end{array}\right) \tag{1.1}$$

1.8 Теоретическая сложность умножения матриц

В рамках такого раздела теории сложности, как мелкозернистая сложность (fine-grained complexity) задача умножения двух матриц оказалась достаточно важной, так как через вычислительную сложность этой задачи можно оценить сложность большого класса различных задач. С примерами таких задач можно ознакомиться в работе [76]. Поэтому рассмотрим алгоритмы нахождения произведения двух матриц более подробно. Далее для простоты мы будем предполагать, что перемножаются две квадратные матрицы одинакового размера $n \times n$.

Для начала построим наивный алгоритм, сконструированный на основе определении произведения матриц. Такой алгоритм представлен на листинге 0. Его работу можно описать следующим образом: для каждой строки в

первой матрице и для каждого столбца в второй матрице найти сумму произведений соответствующих элементов. Данная сумма будет значением соответствующей ячейки результирующей матрицы.

Listing 1 Наивное перемножение матриц

```
1: function MatrixMult(M_1, M_2, G = (S, ⊕, ⊗))
2:     M_3 = empty matrix of size n × n
3:     for i ∈ 0..n − 1 do
4:         for j ∈ 0..n − 1 do
5:             for k ∈ 0..n − 1 do
6:                 M_3[i, j] = M_3[i, j] ⊕ (M_1[i, k] ⊗ M_2[k, j])
7:     return M_3
```

Сложность наивного произведения двух матриц составляет $O(n^3)$ из-за тройного вложенного цикла, где каждый уровень вложенности привносит n итераций. Но можно ли улучшить этот алгоритм? Первый положительный ответ был опубликовал Ф. Штрассен в 1969 году [66]. Сложность предложенного им алгоритма — $O(n^{\log_2 7}) \approx O(n^{2.81})$. Основная идея — рекурсивное разбиение исходных матриц на блоки и вычисление их произведения с помощью только 7 умножений, а не 8.

Рассмотрим алгоритм Штрассена более подробно. Пусть A и B — две квадратные матрицы размера $2^n \times 2^n$ над кольцом $R = (S, \oplus, \otimes)$. Если размер умножаемых матриц не является натуральной степенью двойки, то дополняем исходные матрицы дополнительными нулевыми строками и столбцами. Наша задача найти матрицу $C = A \cdot B$.

Разделим матрицы A, B и C на четыре равные по размеру блока.

$$A = \begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix}, B = \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix}, C = \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

По определению произведения матриц выполняются следующие равенства.

$$\begin{aligned} C_{1,1} &= A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \\ C_{1,2} &= A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2} \\ C_{2,1} &= A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \\ C_{2,2} &= A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2} \end{aligned}$$

Данная процедура не даёт нам ничего нового с точки зрения вычислительной сложности. Но мы можем двинуться дальше и определить следующие элементы.

$$
\begin{aligned}
P_1 &\equiv (A_{1,1} + A_{2,2}) \cdot (B_{1,1} + B_{2,2}) \\
P_2 &\equiv (A_{2,1} + A_{2,2}) \cdot B_{1,1} \\
P_3 &\equiv A_{1,1} \cdot (B_{1,2} - B_{2,2}) \\
P_4 &\equiv A_{2,2} \cdot (B_{2,1} - B_{1,1}) \\
P_5 &\equiv (A_{1,1} + A_{1,2}) \cdot B_{2,2} \\
P_6 &\equiv (A_{2,1} - A_{1,1}) \cdot (B_{1,1} + B_{1,2}) \\
P_7 &\equiv (A_{1,2} - A_{2,2}) \cdot (B_{2,1} + B_{2,2})
\end{aligned}
$$

Используя эти элементы мы можем выразить блоки результирующей матрицы следующим образом.

$$
\begin{aligned}
C_{1,1} &= P_1 + P_4 - P_5 + P_7 \\
C_{1,2} &= P_3 + P_5 \\
C_{2,1} &= P_2 + P_4 \\
C_{2,2} &= P_1 - P_2 + P_3 + P_6
\end{aligned}
$$

При таком способе вычисления мы получаем на одно умножение подматриц меньше, чем при наивном подходе. Это и приводит, в конечном итоге, к улучшению сложности всего алгоритма, который основывается на рекурсивном повторении проделанной выше процедуры.

Впоследствии сложность постепенно понижалась в ряде работ, таких как [51, 15, 58, 20, 21]. Было введено специальное обозначение для показателя степени в данной оценке: ω. То есть сложность умножения матриц — это $O(n^\omega)$, и задача сводится к уменьшению значения ω. В настоящее время работа над уменьшением показателя степени продолжается и сейчас уже предложены решения с $\omega < 2.373$[12].

Всё тем же Ф. Штрассеном ещё в 1969 году была выдвинута гипотеза о том, что для достаточно больших n существует алгоритм, который для любого сколь угодно маленького наперёд заданного ε перемножает матрицы за $O(n^{2+\varepsilon})$. На текущий момент ни доказательства, ни опровержения этой гипотезы не предъявлено.

[12] В данной области достаточно регулярно появляются новые результаты, дающие сравнительно небольшие, в терминах абсолютных величин, изменения. Так, в 2021 была представлена работа, улучшающая значение ω в пятом знаке после запятой [3]. Несмотря на кажущуюся несерьёзность результата, подобные работы имеют большое теоретическое значение, так как улучшают наше понимание исходной задачи и её свойств.

Важной особенностью указанного выше направления улучшения алгоритмов является то, что оно допускает использования (и даже основывается на использовании) более богатых алгебраических структур, чем требуется для определения умножения двух матриц. Так, уже алгоритм Штрасеена использует операцию вычитания, что приводит к необходимости иметь обратные элементы по сложению, а значит определять матрицы над кольцом. Хотя для исходного определения (1.7.9) достаточно более бедной структуры. При этом, часто, структуры, возникающие в прикладных задачах кольцами не являются. Примерами могут служить тропическое (или $\{min, +\}$) полукольцо, играющее ключевую роль в тропической математике, или булево ($\{\vee, \wedge\}$) полукольцо, возникающее, например, при работе с отношениями[13]. Значит, описанные выше решения не применимы и вопрос о существовании алгоритма с менее чем кубической сложностью снова актуален.

В попытках ответить на этот вопрос появились так называемые комбинаторные алгоритмы умножения матриц[14]. Классический результат в данной области — это алгоритм четырёх русских, предложенный В. Л. Арлазаровым, Е. А. Диницем, М. А. Кронродом и И. А. Фараджевым в 1970 году [86], позволяющий перемножить матрицы над конечным полукольцом за $O(n^3/\log n)$. Лучшим результатом[15] в настоящее время является алгоритм со сложностью $\hat{O}(n^3/\log^4 n)$[16] [80].

Как видим, особенности алгебраических структур накладывают серьёзные

[13]Вообще говоря, в некоторых прикладных задачах возникают структуры, не являющиеся даже полукольцом. Предположим, что есть три различных множества S_1, S_2 и S_3 и две двухместные функции $f : S_1 \times S_2 \to S_3$ и $g : S_3 \times S_3 \to S_3$. Этого достаточно, чтобы определить произведение двух матриц M_1 и M_2, построенных из элементов множеств S_1 и S_2 соответственно. Результирующая матрица будет состоять из элементов S_3. Как видно, функции не являются бинарными операциями в смысле нашего определения. Несмотря на кажущуюся экзотичность, подобные структуры возникают на практике при работе с графами и учитываются, например, в стандарте GraphBLAS (`https://graphblas.github.io/`), где, кстати, называются полукольцами, что выглядит не вполне корректно.

[14]В противовес описанным выше, не являющимся комбинаторными. Стоит отметить, что строгое определение комбинаторных алгоритмов отсутствует, хотя этот термин и получил широкое употребление. В частности, Н. Бансал (Nikhil Bansal) и Р. Уильямс (Ryan Williams) в работе [8] дают определение комбинаторного алгоритма, но тут же замечают следующее: “We would like to give a definition of “combinatorial algorithm”, but this appears elusive. Although the term has been used in many of the cited references, nothing in the literature resembles a definition. For the purposes of this paper, let us think of a “combinatorial algorithm” simply as one that does not call an oracle for ring matrix multiplication.”. Ещё один вариант определения и его обсуждение можно найти в [24].

[15]В работе [24] предложен алгоритм со сложностью $\Omega(n^{7/3}/2^{O(\sqrt{\log n})})$, однако авторы утверждают, что сами не уверены в комбинаторности предложенного решения. По-видимому, полученные результаты ещё должны быть проверены сообществом.

[16]Нотация $\hat{O}$ скрывает $poly(\log\log)$ коэффициенты.

ограничения на возможности конструирования алгоритмов. Отметим, что, хотя, в указанных случаях и предлагаются решения лучшие, чем наивное кубическое, они обладают принципиально разной асимптотической сложностью. в первом случае сложность оценивается полиномом, степень которого меньше третьей. Такие решения принято называть *истинно субкубическими* (truly subcubic). В то время как в случае комбинаторных алгоритмов степень полинома остается прежней, третьей, хотя сложность и уменьшается на логарифмический фактор. Такие решения принято называть *слегка субкубическими* (mildly subcubic). Естественный вопрос о существовании истинно субкубического алгоритма перемножения матриц над полукольцами (или же комбинаторного перемножения матриц) всё ещё не решён[17].

[17]Один из кандидатов — работа [24], однако на текущий момент предложенное в ней решение требует проверки.

Глава 2

Некоторые сведения из 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. Кроме этого, поговорим о связи между линейной алгеброй и некоторыми задачами анализа графов. Всё это понадобится нам при последующей работе.

2.1 Основные определения

Определение 2.1.1. *Помеченный ориентированный 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 на рёбрах. В некоторых случаях метки называют *весами*[1] и тогда говорят о *взвешенном* графе.

Определение 2.1.2. В случае, если для любого ребра (u, l, v) в графе также содержится ребро (v, l, u), говорят, что граф *неориентированный*.

В дальнейшем речь будет идти о конечных ориентированных помеченных графах. Мы будем использовать термин *граф* подразумевая именно конечный ориентированный помеченный граф, если только не оговорено противное.

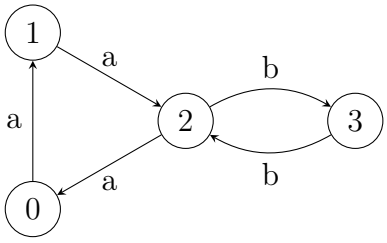
Также мы будем считать, что все вершины занумерованы подряд с нуля. То есть можно считать, что V — это отрезок $[0, |V|-1]$ неотрицательных целых чисел, где $|V|$ — мощность множества V.

[1]Весами метки называют, как правило, тогда, когда они берутся из какого-либо поля, например $\mathbb{R}$ или $\mathbb{N}$.

Пример 2.1.1 (Пример графа и его графического представления)**.** Пусть дан граф

$$\begin{aligned}\mathcal{G} = \langle V &= \{0,1,2,3\},\\ E &= \{(0,a,1),(1,a,2),(2,a,0),(2,b,3),(3,b,2)\},\\ L &= \{a,b\}\rangle.\end{aligned}$$

Графическое представление графа $\mathcal{G}$:

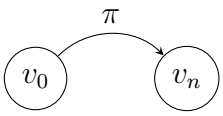


Пример 2.1.2 (Пример рёбер графа)**.** $(0,a,1)$ и $(3,b,2)$ — это рёбра графа $\mathcal{G}_1$. При этом $(3,b,2)$ $(2,b,3)$ — это разные рёбра.

Определение 2.1.3. *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 $v_0 \pi v_n$. Иными совами,

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

Часто для представления пути мы буем использовать следующие нотации:



или

$$v_0 \xrightarrow{l_0} v_1 \xrightarrow{l_1} v_2 \xrightarrow{l_2} \ldots \xrightarrow{l_{n-2}} v_{n-1} \xrightarrow{l_{n-1}} v_n.$$

Пример 2.1.3 (Пример путей графа)**.** $(0,a,1),(1,a,2) = 0\pi_1 2$ — 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 $0\pi_1 2$.

Кроме того, нам потребуется отношение, отражающее факт существования пути между двумя вершинами.

Определение 2.1.4. *Отношение достижимости* в графе: $(v_i, v_j) \in P \iff \exists v_i \pi v_j$.

Отметим, что в некоторых задачах удобно считать по умолчанию, что $(v_i, v_i) \in P$, однако наше определение такого не допускает. Исправить ситуацию можно явно добавив петли (v_i, l, v_i) для всех вершин.

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

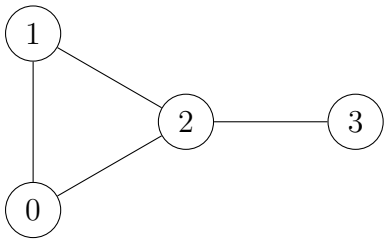
Определение 2.1.5. *Матрица смежности* графа $\mathcal{G} = \langle V, E, L \rangle$ — это квадратная матрица M размера $n \times n$, где $|V| = n$, построенная над коммутативным моноидом $\mathbb{G} = (S, \circ\colon S \times S \to S)$, который конструируется следующим образом.

1. $L \subseteq S$.
2. $\circ$ — коммутативная бинарная операция.
3. Существует $\mathbb{0} \in (S \setminus L)$ — нейтральный элемент относительно $\circ$.

При этом $M[i, j] = \bigcirc_{(i,l,j) \in E} l$, где $\bigcirc_{\varnothing} = \mathbb{0}$.

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

Пример 2.1.4 (Пример матрицы смежности неориентированного графа)**.** Пусть дан следующий неориентированный граф.



$\mathbb{G} = (S, \circ)$ в этом случае конструируется следующим образом. Во-первых, придётся предположить, что L — множество с одним элементом, скажем s, и

считать, что все рёбра помечены им[2]. Далее, $S = L \cup \{n\} = \{s, n\}$, где n — нейтральный элемент относительно $\circ$. Тогда $\circ$ можно определить поточечно следующим образом.

- $s \circ s = s$
- $s \circ n = n \circ s = s$
- $n \circ n = n$

Таким образом, матрица смежности данного графа выглядит следующим образом:

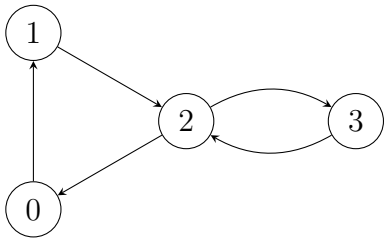
$$\begin{pmatrix} n & s & s & n \\ s & n & s & n \\ s & s & n & s \\ n & n & s & n \end{pmatrix}$$

, что может показаться несколько непривычным. Однако заметим, что построенная нами структура $\mathbb{G} = (\{s, n\}, \circ)$ изоморфна $\mathbb{G}' = (\{1, 0\}, \vee)$. При переходе к $\mathbb{G}'$ мы получим привычную нам булеву матрицу смежности:

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

Заметим, что матрица смежности неориентированного графа всегда симметрична относительно главной диагонали.

Пример 2.1.5 (Пример матрицы смежности ориентированного графа)**.** Дан ориентированный граф:

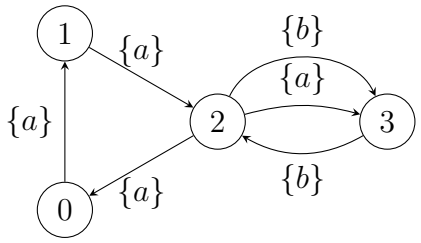


[2]А раз все рёбра имеют одинаковый заранее известный вес, то можно его и не писать для каждого ребра при задании графа. Поэтому привычное нам изображение получается достаточно логичным.

Построить его булеву матрицу смежности можно применив рассуждения из предыдущего примера (2.1.4) и выглядеть она будет следующим образом:

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

Пример 2.1.6 (Пример матрицы смежности помеченного графа)**.** Пусть дан следующий помеченный граф.

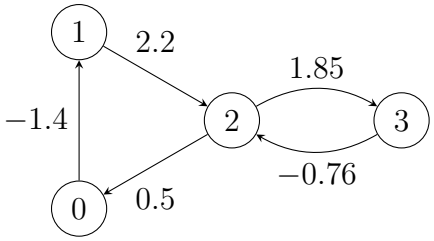


В данном случае $L = \{\{a\}, \{b\}\}$, а $\mathbb{G} = (\{\{a\}, \{b\}, \{a, b\}, \varnothing\}, \cup)$, где $\varnothing$ — нейтральный элемент. Тогда матрица матрица смежности исходного графа выглядит следующим образом:

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

Необходимо заметить, что свойства структуры $\mathbb{G}$, а значит и детали её построения, зависят от задачи, в рамках которой рассматривается граф. В примерах выше мы строили $\mathbb{G}$ из некоторых общих соображений, не специфицируя решаемую задачу, стараясь получить ожидаемый результат. Далее мы рассмотрим пример, в котором видно, как решаемая задача влияет на построение $\mathbb{G}$.

Пример 2.1.7 (Пример матрицы смежности взвешенного графа)**.** Пусть дан следующий взвешенный граф:



Будем считать, что веса берутся из $\mathbb{R}$, а решаемая задача — поиск кратчайших путей между вершинами. В таком случае естественно предположить, что для любой вершины v_i существует петля $(v_i, 0, v_i)$, хоть она явно и не изображена. Далее, $\mathbb{G} = (\mathbb{R} \cup \{\infty\}, \min)$, где ∞ — нейтральный элемент относительно операции min.

В результате мы получим следующую матрицу смежности:

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

Таким образом, уже можно заметить, что введение моноида как абстракции позволяет достаточно унифицированным образом смотреть на различные графы и их матрицы смежности. Далее мы увидим, что данный путь позволит решать унифицированным образом достаточно широкий круг задач, связанных с анализом путей в графах. Но сперва мы сформулируем различные варианты задачи поиска путей в графе.

2.2 Обход графа в ширину

Обход графа в ширину — это одна из фундаментальных задач анализа графов, для решения которой существует соответствующий алгоритм, изложенный в классической литературе (см., например, [] или []).

В общих чертах, задача заключается в том, чтобы начиная с некоторой вершины графа (источника) обойти все достижимые из неё вершины в некотором порядке. Шаг — просмотр все смежных. И так для всего фронта. Главное не посещать одну и ту же вершину несколько раз.

Псевдокод классического алгоритма

Пример.

Алгоритм обхода в ширину может быть переформулирован в терминах матрично-векторных операций. Решение через линейную алгебру.

Псевдокод алгоритма на ЛА

Пример.

multiple-source BFS. Тот же обход в ширину, только источников несколько и надо помнить, какая из вершин из какого источника достижима. Постановка задачи. Решение через линейную алгебру [30] Уже не вектор, а матрица: храним информацию про каждую стартовую вершину отдельно.

Псевдокод алгоритма на ЛА

Пример.

2.3 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, постановки различаются тем, что именно мы хотим получить в качестве результата. Здесь наиболее частыми являются следующие варианты.

- Наличие хотя бы одного пути, удовлетворяющего ограничениям, в графе. В данном случае не важно, между какими вершинами существует путь, важно лишь наличие его в графе.
- Наличие пути, удовлетворяющего ограничениям, между некоторыми вершинами: задача достижимости. При данной постановке задачи, нас интересует ответ на вопрос достижимости вершины v_i из вершины v_j по пути, удовлетворяющему ограничениям. Такая постановка требует лишь проверить существование пути, но не его предоставления в явном виде.
- Поиск одного пути, удовлетворяющего ограничениям: необходимо не только установить факт наличия пути, но и предъявить его. При этом часто подразумевается, что возвращается любой путь, являющийся решением, без каких-либо дополнительных ограничений. Хотя, например, в некоторых задачах дополнительное требование простоты или наименьшей длины выглядит достаточно естественным.
- Поиск всех путей: необходимо предоставить все пути, удовлетворяющие заданным ограничениям.

С другой стороны, задачи могут различаться ещё и тем, как фиксируются множества стартовых и конечных вершин. Здесь возможны следующие варианты:

- от одной вершины до всех,
- между всеми парами вершин,
- между фиксированной парой вершин,

- между двумя множествами вершин V_1 и V_2, что подразумевает решение задачи для всех $(v_i, v_j) \in V_1 \times V_2$.

Стоит отметить, что последний вариант является самым общим и остальные — лишь его частные случаи. Однако этот вариант часто выделяют отдельно, подразумевая, что остальные, выделенные, варианты в него не включаются.

В итоге, перебирая возможные варианты желаемого результата и способы фиксации стартовых и финальных вершин, мы можем сформулировать достаточно большое количество задач. Например, задачу поиска всех путей между двумя заданными вершинами, задачу поиска одного пути от фиксированной стартовой вершины до каждой вершины в графе, или задачу достижимости между всеми парами вершин.

Часто поиск путей сопровождается изучением их свойств, что далее приводит к формулированию дополнительных ограничений на пути в терминах этих свойств. Например, можно потребовать, чтобы пути были простыми или не проходили через определённые вершины. Один из естественных способов описывать свойства и, как следствие, задавать ограничения — это использовать ту алгебраическую структуру, из которой берутся веса рёбер графа[3].

Предположим, что дан граф $\mathcal{G} = \langle V, E, L \rangle$, где $L = \langle S, \oplus, \otimes \rangle$ — это полукольцо. Тогда изучение свойств путей можно описать следующим образом:

$$\{(v_i, v_j, c) \mid \exists v_i \pi v_j, c = \bigoplus_{\forall v_i \pi v_j} \bigotimes_{(u,l,v) \in \pi} l\}. \tag{2.1}$$

Иными словами, для каждой пары вершин, для которой существует хотя бы один путь, их соединяющий, мы агрегируем (с помощью операции $\oplus$ из полукольца) информацию обо всех путях между этими вершинами. При этом информация о пути получается как свёртка меток рёбер пути с использованием операции $\otimes$[4].

Естественным требованием (хотя бы для прикладных задач, решаемых таким способом) является существование и конечность указанной суммы. На

[3]На самом деле здесь наблюдается некоторая двойственность. С одной стороны, действительно, удобно считать, что свойства описываются в терминах некоторой заданной алгебраической структуры. Но, вместе с этим, структура подбирается исходя из решаемой задачи.

[4]Заметим, что детали свёртки вдоль пути зависят от свойств полукольца (и от решаемой задачи). Так, если полукольцо коммутативно, то нам не обязательно соблюдать порядок рёбер. В дальнейшем мы увидим, что данные особенности полукольца существенно влияют на особенности алгоритмов решения соответствующих задач.

данном этапе мы не будем касаться того, какие именно свойства полукольца могут нам обеспечить данное свойство, однако в дальнейшем будем считать, что оно выполняется. Более того, будем стараться приводить частные для конкретной задачи рассуждения, показывающие, почему это свойство выполняется в рассматриваемых в задаче ограничениях.

Описанная выше задача общего вида называется анализом свойств путей алгебраическими методами (Algebraic Path Problem [9]) и предоставляет общий способ для решения широкого класса прикладных задач[5]. Наиболее известными являются такие задачи, как построение транзитивного замыкания графа и поиск кратчайших путей (All PAirs Shortest Path или APSP). Далее мы подробнее обсудим эти две задачи и предложим алгоритмы их решения.

2.4 Алгоритм Флойда-Уоршелла

Наиболее естественным образом решение обсуждаемых выражается в терминах операций над матрицей смежности исходного графа. Поэтому предположим, что исходный граф задан матрицей над моноидом $\mathbb{G} = (S, \oplus)$.

Как мы видели ранее, операция $\oplus$ позволяет нам агрегировать информацию по всем параллельным рёбрам. Ровно она же и будет агрегировать информацию по всем путям между двумя вершинами[6]. Таким образом, осталось сконструировать операцию, отвечающую за агрегацию информации вдоль пути. Здесь мы будем исходить из того, что новый путь может быть получен из двух подпутей, а свойство нового пути зависит только от свойств исходных подпутей.

Таким образом, дополнительная операция, обозначим её $\otimes : S \times S \to S$[7], должна вести себя следующим образом. Пусть S — носитель моноида, $\mathbb{0} \in S$ — нейтральный элемент относительно $\oplus$.

[5] В работе “Path Problems in Networks” [9] собран действительно большой список прикладных задач с описанием соответствующих полуколец. Сводная таблица на страницах 58–59 содержит 29 различных прикладных задач и соответствующих полуколец.

[6] Вообще говоря, работая с матрицей смежности мы не видим разницу между путём и ребром, так как любая запись в матрице смежности в ячейке $[i, j]$ говорит нам только о том, что вершины i и j связаны и эта связь обладает некоторым свойством (значение в ячейке), и ничего не говорит о том, как эта связь устроена.

[7] При первом рассмотрении такой выбор кажется контринтуитивным. Действительно, ведь при соединении путей мы как бы “складываем” их веса. Но при более детальном анализе поведения этой операции, в частности, относительно нейтрального элемента, становится понятно, что она ведёт себя очень похоже на умножение. Вероятно, стоит обратить внимание на операцию конкатенации, которая, с одной стороны, “делает то, что нам нужно”, а с другой, (и неспроста) часто обозначается $\cdot$.

- $s_1 \otimes s_2 = s_3, s_i \in S, s_i \neq \mathbb{0}$: если существует путь $i\pi j$ со свойством s_1 и путь $j\pi k$ со свойством s_2, то существует путь $i\pi k$ со свойством s_3.

- $s \otimes \mathbb{0} = \mathbb{0}$: если существует путь $i\pi j$ со свойством s и не существует пути $j\pi k$, то не существует и пути $i\pi k$.

- $\mathbb{0} \otimes s = \mathbb{0}$: если не существует пути $i\pi j$ и существует путь $j\pi k$ со свойством s, то не существует и пути $i\pi k$.

- $\mathbb{0} \otimes \mathbb{0} = \mathbb{0}$: если не существует пути $i\pi j$ и не существует пути $j\pi k$, то не существует и пути $i\pi k$.

Новую операцию добавим к моноиду и получим новую алгебраическую структуру $\mathbb{G}' = (S, \oplus, \otimes)$. Данная структура является коммутативным моноидом по сложению (по построению) с нейтральным элементом $\mathbb{0}$. Из построения $\otimes$ видно, что $\mathbb{0}$ является аннигилятором. Ничего более про операцию $\otimes$, а значит и про $\mathbb{G}'$ мы сказать, исходя из построения, не можем. Классический фреймворк для решения algebraic path problem подразумевает, что $\mathbb{G}'$ является полукольцом, однако далее мы увидим, что существуют задачи, в которых $\otimes$, например, не является ассоциативной[8]. А значит, согласно нашему определению, $\mathbb{G}'$ полукольцом не является.

Теперь, когда построена алгебраическая структура, обеспечивающая вычисление формулы 2.1, мы можем предложить алгоритм вычисления этой формулы и данным алгоритмом в интересующих нас частных случаях будет являться алгоритм Флойда-Уоршелла [31, 57, 75]. Псевдокод алгоритма представлен на листинге 2, а его сложность $O(n^3)$. Он практически дословно основан на описанной выше идее сборки путей из двух подпутей: тройной вложенный цикл перебирает все возможные разбиения пути на две части, а в строке 7 как раз и происходит вычисление формулы 2.1.

Необходимо обратить внимание на несколько вещей. Первая — порядок обхода. Внешний цикл перебирает возможные точки разбиения (хотя мог бы, например, перебирать начальные вершины) для того, чтобы гарантировать правильный порядок вычисления подпутей (информация ни о каких подпутях не будет получена после того, как они поучаствовали в построении более длинного пути). Вторая — количество итераций. В данном случае мы ограничились тройным вложенным циклом от 0 до n и для наших задач этого будет достаточно, однако, как доказательство этого факта, так и построение

[8]Такой будет рассматриваемая в данной работе задача достижимости с ограничениями в терминах формальных языков. Другие примеры можно найти в уже упоминавшейся работе [9]

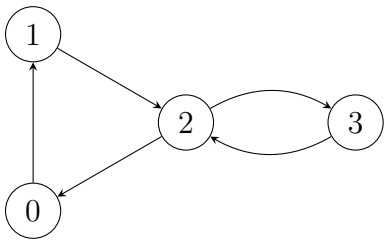
аналогичного алгоритма для других задач требует аккуратного анализа решаемой задачи и последующего доказательства корректности построенного алгоритма.

Listing 2 Алгоритм Флойда-Уоршелла

```
1: function FloydWarshall(𝒢)
2:     M ← матрица смежности 𝒢                ▷ Матрица над 𝔾 = (S, ⊕, ⊗)
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] ← M[i, j] ⊕ (M[i, k] ⊗ M[k, j])
8:     return M
```

Хотя изначально данный алгоритм был предложен для решения задачи о кратчайших путях, при абстрагировании алгебраической структуры он превращается в алгоритм решения целого ряда задач. В частности — нахождения транзитивного замыкания. Так, если возьмём тропическое полукольцо $(\mathbb{R}_{+\infty}, \min, +)$, то получим алгоритм для поиска кратчайших путей. Если же возьмём булево полукольцо, то получим алгоритм для построения транзитивного замыкания графа.

Пример 2.4.1 (Транзитивное замыкание графа)**.** Пусть дан следующий граф:



Его матрица смежности:

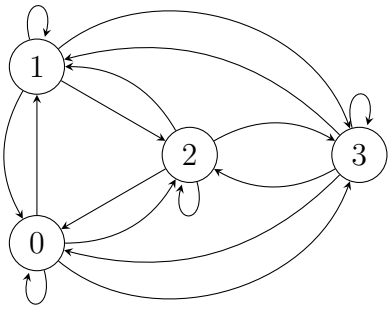
$$M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Здесь мы считаем, что отношение достижимости не рефлексивно: все диагональные элементы матрицы M равны 0.

Воспользовавшись алгоритмом из листинга 2, специализированного на случай булева полукольца, можно получить следующую матрицу смежности.

$$M' = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

А значит, транзитивным замыканием исходного графа является полный граф и он выглядит следующим образом.



Заметим, что рефлексивность отношения (значения элементов на главной диагонали матрицы смежности) непосредственно связана с особенностями решаемой задачи. Например, если говорить о кратчайших расстояниях, то кажется естественным считать расстояние от вершины до самой себя равной нулю. Однако для задачи транзитивного замыкания это уже не столь естественное предположение и часто отдельно говорят о рефлексивно-транзитивном замыкании, которое отдельно явным образом привносит рефлексивность (петли, диагональные элементы матрицы смежности).

Аналогичным образом, используя данный алгоритм, но уже для тропического полукольца, можно решить задачу о поиске кратчайших путей для графа из примера 2.1.7.

Идеи, заложенные в алгоритме Флойда-Уоршелла, а также возможность абстрагировать его, помогут нам в дальнейшем предложить алгоритм для задачи достижимости с ограничениями в терминах формальных языков.

2.5 Анализ путей в графе и линейная алгебра

В данной главе мы рассмотрим некоторые связи[9] между графами и операциями над ними и матрицами и операциями над матрицами.

Как мы видели, достаточно естественное представление графа — это его матрица смежности. Далее можно заметить некоторое сходство между определением матричного умножения 1.7.9 и мыслями, которыми мы руководствовались, вводя операцию $\otimes$ (2.4). Действительно, пусть есть матрица M размера $n \times n$ над $mathbbG = (S, \oplus, \otimes)$[10] — матрица смежности графа. Умножим её саму на себя: вычислим $M' = M \cdot M$. Тогда, по 1.7.9, $M'[i,j] = \bigoplus_{0 \le l < n} M[i,l] \otimes M[l,j]$. Размер M' также $n \times n$. То есть M' задаёт такой граф, что в нём будет путь со свойством, являющимся агрегацией свойств всех путей, составленных из двух подпутей в исходном графе. А именно, если в исходном графе есть путь из i в l с некоторым свойством (его значение хранится в $M[i,l]$), и был путь из l в j (его значение хранится в $M[l,j]$), то в новом графе свойство $M[i,l] \otimes M[l,j]$ аддитивно (используя $\oplus$) учтётся в свойстве пути из i в j.

Таким образом, произведение матриц смежности соответствует обработке информации о путях интересующим нас образом. Это наблюдение позволяет предложить решение задач анализа свойств путей, основанное на операциях над матрицами. Рассмотрим такое решение для задачи о кратчайших путях.

Пусть D_k — матрица кратчайших путей, состоящих не более чем из k рёбер. То есть $D_k[i,j]$ — это длина кратчайшего пути из вершины i в вершину j, такого, что он состоит не более чем из k ребер. Если такого пути нет, то $D_k[i,j] = \infty$.

Таким образом, $D_1 = M$, где M — это матрица смежности исходного графа, а решением APSP является D_{n-1}, вычисляемая с помощью следующего рекуррентного соотношения:

$$D(1) = M$$

[9] Связь между графами и линейной алгеброй — обширная область, в которой можно даже выделить отдельные направления, такие как спектральная теория графов. С точки зрения практики данная связь также подмечена давно и более полно с ней можно ознакомиться, например, в работах [43, 25]. Кроме этого, много полезной информации можно найти на сайте `https://graphblas.github.io/GraphBLAS-Pointers/`.

[10] Здесь мы уже сталкиваемся с тем, что могут иметь смысл относительно экзотические алгебраические структуры. Действительно, как мы выяснили, матрица смежности может быть определена на чем-то более бедным, чем полукольцо, а матичное умножение мы определяли над полукольцом. Но если задуматься, то и для определения произведения матриц полукольцо вовсе необязательно, достаточно более бедной структуры.

$$D(2) = D(1) \cdot M = M^2$$
$$D(3) = D(2) \cdot M = M^3$$
$$\dots$$
$$D(n-1) = D(n-2) \cdot M = M^{(n-1)}$$

Здесь мы пользуемся той особенностью задачи, что кратчайший путь в ориентированном графе (без отрицательных циклов) не может быть длиннее n[11]. Более того, мы пользуемся тем, что используемая структура именно полукольцо, а исходное отношение рефлексивно[12].

Таким образом, решение APSP сведено к произведению матриц над тропическим полукольцом, однако вычислительная сложность решения слишком большая: $O(nK(n))$, где $K(n)$ — сложность алгоритма умножения матриц.

Чтобы улучшить сложность, заметим, что, например, D_3 вычислять не обязательно, так как можно сразу вычислить D_4 как $D_2 \cdot D_2$.

В итоге получим следующую последовательность вычислений:

$$D_1 = M$$
$$D_2 = M^2 = M \cdot M$$
$$D_4 = M^4 = M^2 \cdot M^2$$
$$D_8 = M^8 = M^4 \cdot M^4$$
$$\dots$$
$$D_{2^{\log(n-1)}} = M^{2^{\log(n-1)}} = M^{2^{\log(n-1)}-1} \cdot M^{2^{\log(n-1)}-1}$$
$$D_{n-1} = D_{2^{\log(n-1)}}$$

Теперь вместо n итераций нам нужно $\log n$, а итоговая сложность решения — $O(\log nK(n))$[13]. Данный алгоритм называется *repeated squaring*[14]. Здесь мы предполагаем, что $n-1 = 2^k$ для какого-то k. На практике такое верно далеко

[11] Раз отрицательных циклов нету, то проходить через одну вершину, коих n, больше одного раза бессмысленно.

[12] Тот факт, что в любой вершине есть петля с весом 0, а 0 — нейтральный для $\otimes$, позволяет не суммировать матрицы. Итоговая матрица содержит данные о путях длины *не больше чем* вычисляемая степень, хотя должна бы содержать данные о путях ровно и только такой длины. Предлагается самостоятельно исследовать данный феномен.

[13] Заметим, что это оценка для худшего случая. На практике при использовании данного подхода можно прекращать вычисления как только при двух последовательных шагах получились одинаковые матрицы ($D_i = D_{i-1}$). Это приводит нас к понятию *неподвижной точки*, обсуждение которого лежит за рамками повествования.

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

не всегда. Если это условие не выполняется, то необходимо взять ближайшую сверху степень двойки[15].

Таким образом, APSP сводится к умножению матриц над полукольцом, что, к сожалению, не позволяет этим путём получить истинно субкубический алгоритм для задачи. тем не менее, позволяет получить слегка субкубический. Приведем некоторые работы по APSP для ориентированных графов с вещественными весами (здесь n — количество вершин в графе), по которым можно более детально ознакомиться как с историей вопроса, так и с текущими результатами:

- M.L. Fredman (1976) — $O(n^3(\log\log n/\log n)^{\frac{1}{3}})$ — использование дерева решений [32]
- 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 [67]
- Y. Han (2004) — $O(n^3(\log\log n/\log n)^{\frac{5}{7}})$ [34]
- T. Takoaka (2004) — $O(n^3(\log\log n)^2/\log n)$ [68]
- T. Takoaka (2005) — $O(n^3\log\log n/\log n)$ [69]
- U. Zwick (2004) — $O(n^3\sqrt{\log\log n}/\log n)$ [85]
- T.M. Chan (2006) — $O(n^3/\log n)$ — многомерный принцип “разделяй и властвуй” [17]

Вопрос же о истинно субкубических алгоритмах решения APSP всё ещё открыт [18] и активно обсуждается в научном сообществе.

Аналогичным образом можно свести транзитивное замыкание к матричным операциям. Предлагаем проделать это самостоятельно, заодно обратив внимание на важность рефлексивности (в примере 2.4.1 её нет).

Заметим, что оптимизация, связанная с возведением в квадрат возможна только при ассоциативности произведения матриц, что зависит от свойств алгебраической структуры, над которой построены матрицы. Для полукольца такая оптимизация законна, однако в некоторых случаях её применить нельзя. Таким случаем как раз и будет рассматриваемая в нашей работе задача.

[15]Кажется, что это приведёт к избыточным вычислениям. Попробуйте оценить, на сколько много лишних вычислений будет сделано в худшем случае.

Поэтому построение алгоритма её решения через операции над матрицами, хотя и будет основано на указанных выше соображениях, но будет сопряжено с некоторыми трудностями.

Глава 3

Общие сведения теории формальных языков[1]

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

Определение 3.0.1. *Алфавит* — это конечное множество. Элементы этого множества будем называть *символами*.

Пример 3.0.1. Примеры алфавитов

- Латинский алфавит $\Sigma = \{a, b, c, \ldots, z\}$
- Кириллический алфавит $\Sigma = \{$а, б, в, ..., я$\}$
- Алфавит натуральных 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: $a, b, \ldots, x, y, z$.

[1] В рамках данной работы мы будем говорить о "типичных" языках, элементами которых являются объекты, максимально похожие на строки. При этом будет оставлен за бортом тот факт, что базовое определение позволяет нам рассматривать в качестве "строительных элементов" (алфавита) практически произвольные объекты, а значит, создавать весьма нетривиальные конструкции в качестве слов языка. Примерами "нестроковых" языков могут послужить языки деревьев [19] или языки графов [28, 22].

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 3.0.2. *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 $a_i \in \Sigma$ для любого i.

Определение 3.0.3. 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 3.0.4. *Язык* над алфавитом Σ — это множество слов над алфавитом Σ.

Пример 3.0.2. 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 универсального множества Σ^* — множества всех слов над алфавитом Σ.

Заметим, что язык не обязан быть конечным множеством, в то время как алфавит в нашей области всегда конечен[2] и изучаем мы конечные слова[3].

Можно выделить следующие основные *способы задания языков.*

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

[2]Существуют ситуации, когда возникают бесконечные алфавиты.

[3]Существуют ситуации, когда возникают бесконечные слова. Например работы по обработке потоков.

Общие слова про порождающие грамматики. Через машины Маркова, переписывания. Далее — от того, какие ограничения на правила машины, зависит класс языков.

Основные классы языков.

Пару слов про то, что через переписывания не всегда удобно, не всегда работает. Булевы грамматики.

Теоретико-множественные задачи над языками и их применение. О том, что многое — про пересечение, проверку пустоты, вложенность.

Глава 4

Регулярные языки

Регулярные языки, конечные автоматы, взаимные конвертации, замкнутость.

Определение 4.0.1. Регулярное множество.

4.1 Конечные автоматы

Определение 4.1.1. Конечный автомат.

Пример КА.

Конфигурация, переход между конфигурациями.

Пример интерпретации конечного автомата.

Построение КА по регулярке и регулярки по КА.

Алгоритмы

Примеры.

4.2 Лево(право)линейные грамматики

Определение 4.2.1. Лево (право)линенйная грамматика

Построение грамматики по автомату.

Пример построения грамматики по автомату.

Автомат по грамматике.

4.3 Лемма о накачке

Лемма о накачке для регулярных языков.

Доказательство леммы о накачке для регулярных языков.

4.4 Замкнутость регулярных языков относительно операций

Доказательство замкнутости относительно операций. Алгоритмы для соответствующих операций.

Линейная алгебра для работы с регулярными языками: пересечение, замыкание.

Построение пересечения через тензорное произведение автоматов.

Пересечение через синхронный обход в ширину.

Глава 5

Контекстно-свободные языки и грамматики

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

Определение 5.0.1. *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 (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 5.0.1. 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:

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

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 5.0.2. *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 5.0.3. *Рефлексивно-транзитивное замыкание отношения* — это наименьшее рефлексивное и транзитивное отношение, содержащее исходное.

Определение 5.0.4. *Отношение выводимости* является рефлексивно-транзитивным замыканием отношения непосредственной выводимости; обозначается $\overset{*}{\Rightarrow}$.

- $\alpha \overset{*}{\Rightarrow} \beta$ означает $\exists \gamma_0, \dots \gamma_k \in \{\Sigma \cup N\}^* \cup \varepsilon :\ \alpha \Rightarrow \gamma_0 \Rightarrow \gamma_1 \Rightarrow \dots \Rightarrow \gamma_{k-1} \Rightarrow \gamma_k \Rightarrow \beta$
- 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 5.0.2. Пример вывода цепочки -1101 в грамматике:

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

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

Определение 5.0.5 (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 5.0.6. *Левосторонний вывод.* На каждом шаге вывода заменяется самый левый нетерминал.

Определение 5.0.7. *Правосторонний вывод.* На каждом шаге вывода заменяется самый правый нетерминал.

Пример 5.0.3. Приведем пример левостороннего вывода цепочки *cbaa* в грамматике:

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

Жирным выделен нетерминал, заменяемый на каждом шагу вывода.

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

Аналогично левостороннему можно определить правосторонний вывод.

Определение 5.0.8. *Язык, задаваемый грамматикой* — множество строк, выводимых в грамматике $L(G) = \{\omega \in \Sigma^* \mid S \overset{*}{\Rightarrow} \omega\}$.

Определение 5.0.9. Грамматики G_1 и G_2 называются *эквивалентными*, если они задают один и тот же язык: $L(G_1) = L(G_2)$

Пример 5.0.4. Пример эквивалентных грамматик для языка целых чисел в двоичной системе счисления.

$\Sigma = \{0, 1, -\}$	$\Sigma = \{0, 1, -\}$
$N = \{S, N, A\}$	$N = \{S, A\}$
$S \to 0 \mid N \mid -N$	$S \to 0 \mid 1A \mid -1A$
$N \to 1A$	$A \to 0A \mid 1A \mid \varepsilon$
$A \to 0A \mid 1A \mid \varepsilon$	

Определение 5.0.10. *Неоднозначная грамматика* — грамматика, в которой существует 2 и более левосторонних (правосторонних) выводов для одного слова.

Пример 5.0.5. Неоднозначная грамматика для правильных скобочных последовательностей:

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

Два различных левосторонних вывода строки ()()():

$$S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()SS \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()(S) \Rightarrow ()()()$$

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow (S)SS \Rightarrow ()SS \Rightarrow ()(S)S \Rightarrow ()()S \Rightarrow ()()(S) \Rightarrow ()()()$$

Определение 5.0.11. *Однозначная грамматика* — грамматика, в которой существует не более одного левостороннего (правостороннего) вывода для каждого слова.

Пример 5.0.6. Однозначная грамматика для правильных скобочных последовательностей

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

Определение 5.0.12. *Существенно неоднозначные языки* — языки, для которых невозможно построить однозначную грамматику.

Пример 5.0.7. Пример существенно неоднозначного языка

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

5.1 Дерево вывода

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

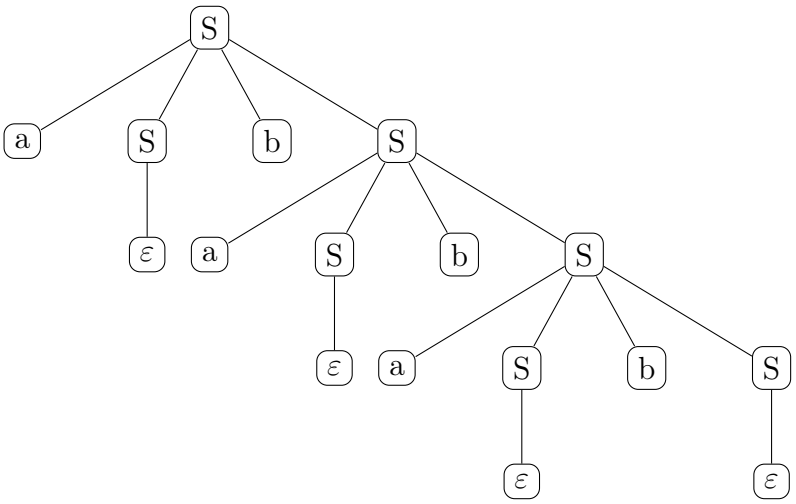
Определение 5.1.1. Деревом вывода цепочки ω в грамматике $G = \langle \Sigma, N, S, P \rangle$ называется дерево, удовлетворяющее следующим свойствам.

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 5.1.1. Построим дерево вывода цепочки *ababab* в грамматике

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



Теорема 5.1.1. Пусть $G = \langle \Sigma, N, P, S \rangle$ — КС-грамматика. Вывод $S \overset{*}{\Rightarrow} \alpha$, где $\alpha \in (N \cup \Sigma)^*, \alpha \neq \varepsilon$ существует $\Leftrightarrow$ существует дерево вывода в грамматике G с кроной α.

5.2 Пустота КС-языка

Теорема 5.2.1. 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 5.2.2. 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.

Вывод цепочки ω имеет следующий вид:

$$S \overset{*}{\Rightarrow} \alpha A_{n_1} \beta \overset{*}{\Rightarrow} \alpha\omega_1\beta; S \overset{*}{\Rightarrow} \alpha\gamma A_{n_2}\delta\beta \overset{*}{\Rightarrow} \alpha\gamma\omega_2\delta\beta \overset{*}{\Rightarrow} \omega,$$

при этом ω_2 является подцепочкой ω_1.

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

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. □

5.3 Нормальная форма Хомского

Определение 5.3.1. 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 $A, B, C \in N$, а стартовый нетерминал S не содержится в правой части правила.
- $A \to a$, где $A \in N, a \in \Sigma$
- $S \to \varepsilon$: только из стартового нетерминала выводима пустая строка.

Теорема 5.3.1. 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 5.3.2. 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, где C_i — новый нетерминал:

$$\begin{aligned} A &\to C_0 C_1 \dots C_k \\ C_0 &\to B_0 \\ C_1 &\to B_1 \\ &\dots \\ C_k &\to B_k \end{aligned}$$

□

Лемма 5.3.3. 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:

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

□

Лемма 5.3.4. Для любой КС-грамматики можно построить эквивалентную, не содержащую ε-правил.

Доказательство. Рекурсивно определим ε-правила:

- $A \to \varepsilon$ — ε-правило
- $A \to B_0 \dots B_k$ — ε-правило, если $\forall i: \ B_i$ — 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 5.3.5. Для любой КС-грамматики можно построить эквивалентную, не содержащую 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 5.3.2. Нетерминал A называется *порождающим*, если из него может быть выведена конечная терминальная цепочка. Иначе он называется *непорождающим*.

Лемма 5.3.6. 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 5.3.1. 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 5.3.3. 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

5.4 Лемма о накачке

Лемма 5.4.1. 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 $k \geq 0$ $uv^k xy^k z \in L$.

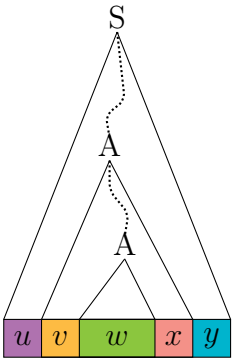


Рис. 5.1: Разбиение цепочки для леммы о накачке

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.

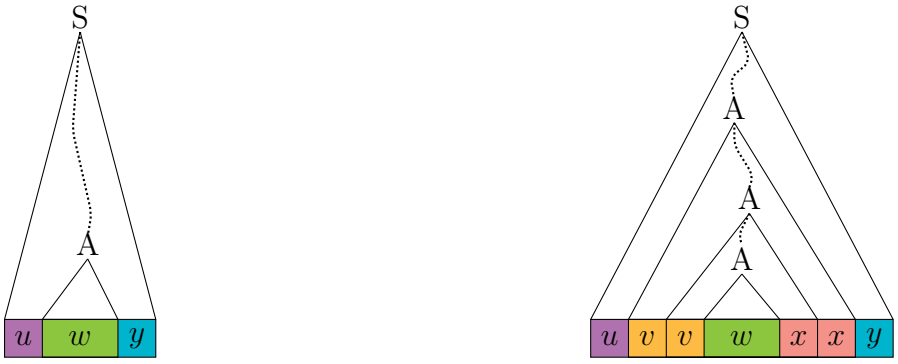
4. Пойдём дальше и найдём второе появление N_1. Цепочка, задаваемая поддеревом для этого узла — это vxy из леммы.

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

Надо только проверить выполнение ограничений на длины.

Нахождение разбиения и пример накачки продемонстрированы на рисунках 5.1 и 5.2.

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



(a) $k = 0$. (b) $k = 2$.

Рис. 5.2: Пример накачки цепочки с рисунка 5.1

5.5 Замкнутость КС языков относительно операций

Теорема 5.5.1. 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 $L_2 = {L_1}^r = \{l^r \mid l \in L_1\}$ является контекстно-свободным.
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 грамматику нового языка, имея грамматики для исходных. Будем предполагать, что множества нетерминальных символов различных грамматик для исходных языков

не пересекаются. Пусть $G_1 = \langle \Sigma_1, N_1, P_1, S_1 \rangle$ — грамматика для L_1, $G_1 = \langle \Sigma_2, N_2, P_2, S_2 \rangle$ — грамматика для L_2.

1. $G_3 = \langle \Sigma_1 \cup \Sigma_2, N_1 \cup N_2 \cup \{S_3\}, P_1 \cup P_2 \cup \{S_3 \to S_1 \mid S_2\}, S_3 \rangle$ — грамматика для L_3.
2. $G_3 = \langle \Sigma_1 \cup \Sigma_2, N_1 \cup N_2 \cup \{S_3\}, P_1 \cup P_2 \cup \{S_3 \to S_1 S_2\}, S_3 \rangle$ — грамматика для L_3.
3. $G_2 = \langle \Sigma_1, N_1 \cup \{S_2\}, P_1 \cup \{S_2 \to S_1 S_2 \mid \varepsilon\}, S_2 \rangle$ — грамматика для L_2.
4. $G_2 = \langle \Sigma_1, N_1, \{N^i \to \omega^R \mid N^i \to \omega \in P_1\}, S_1 \rangle$ — грамматика для $L_2 = L_1^r$.

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 исходный 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 регулярным 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 5.5.2. 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 грамматикой:

$$\begin{aligned} S &\to Sc \\ S &\to T \\ T &\to aTb \\ T &\to Tb \\ T &\to b \end{aligned}$$

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 $L_7 = L_4 \cup L_5 \cup L_6$ — контекстно-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 предыдущего пункта мы знаем, что L_3 не является контекстно-свободным.

□

5.6 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 5.6.1. 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.

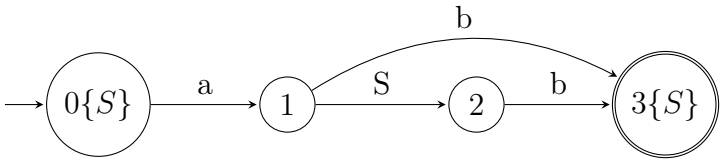
Построение РКА по грамматике.

Допущение о том, что все состояния занумерованы подряд и уникальны.

Немного ссылок на работы по РКА.

Построим рекурсивный автомат для грамматики G:

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



(5.1)

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.

Пример интерпретации конечного автомата.

Глава 6

Многокомпонентные контекстно-свободные языки

Общая теория. Определение, свойства, классы.

Леммы о накачке.

Про MIX и O_n

Глава 7

Задача о поиске путей с ограничениями в терминах формальных языков

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

7.1 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, составленные из меток вдоль них, принадлежат заданному языку.

Определение 7.1.1. *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 [44]), но это уже другая история.

Другим вариантом постановки задачи является задача достижимости.

Определение 7.1.2. *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, контекстно-свободным, или многокомпонентным контекстно-свободным.

Если $\mathcal{L}$ — регулярный, $\mathcal{G}$ можно рассматривать как недетерминированный конечный автомат (НКА), в котором все вершины являются одновременно и стартовыми, и конечными. 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.

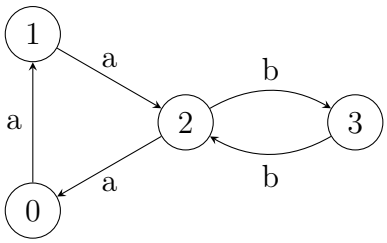
Путь $G = \langle \Sigma, N, P \rangle$ — контекстно-свободная грамматика. Будем считать, что $L \subseteq \Sigma$. Мы не фиксируем стартовый нетерминал в определении грамматики, поэтому, чтобы описать язык, задаваемый ей, нам необходимо отдельно зафиксировать стартовый нетерминал. Таким образом, будем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

Пример 7.1.1. Пример задачи поиска путей.

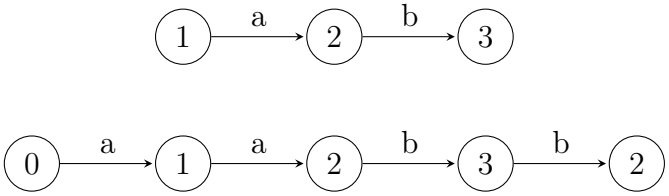
Дана грамматика G, задающая язык $\mathcal{L} = a^n b^n$:

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

И дан граф $\mathcal{G}$:



Кратчайшими путями, принадлежащими множеству $\Pi = \{\pi \mid \omega(\pi) \in \mathcal{L}\}$, являются:



7.2 О разрешимости задачи

Задачи из определения 7.1.1 и 7.1.2 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 7.2.1. 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 [49] 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 [50].

- Ещё один класс языков из альтернативной иерархии, не сравнимой с Иерархией Хомского, — MCFG (multiple context-free grammars) [63]. 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 [63].

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

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

7.3 Области применения

Поиск путей с ограничениями в виде формальных языков широко применяется в различных областях. Ниже даны ключевые работы по применению поиска путей с контекстно-свободными ограничениями и ссылки на них для более детального ознакомления.

- Межпроцедурный Статанализ кода. Идея начала активно разрабатываться Томасом Репсом [55]. Далее последовал ряд, в том числе инженерных работ, применяющих достижимость с контекстно-свободными ограничениями для анализа указателей, анализа алиасов и других прикладных задач [52, 12, 83].
- Графовые БД. Впервые задача сформулирована Михалисом Яннакакисом [79]. Запросы с контекстно-свободными ограничениями нашли своё применение различных областях.
  - Социальные сети [36].
  - RDF обработка [82].
  - Биоинформатика [64].

Глава 8

Поиск путей с регулярными ограничениями

8.1 Достижимость между всеми парами вершин

Через тензорное произведение.

Классическое построение пересечения автоматов строит их тензорное произведение.

Так как мы хотим отвечать ещё и на вопрос о достижимости, что нам надо ещё и транзитивное замыкание посчитать.

8.2 Достижимость с несколькими источниками

Достижимость от нескольких стартовых вершин через обход в ширину, основанный на линейной алгебре [30].

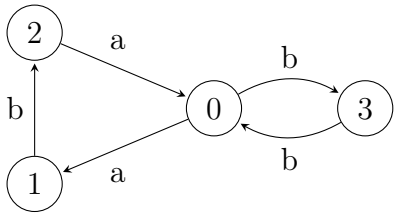
Идея алгоритма основана на одновременном обходе в ширину графа и конечного автомата, построенного по грамматике.

В классической версии обхода в ширину, основанного на линейной алгебре, используется вектор, куда записывается фронт обхода графа. Так, один раз перемножая этот вектор на матрицу смежности графа, можно совершать один шаг в обходе графа. Покажем на примере, как данный метод может быть использован, когда мы накладываем дополнительные ограничения в виде регулярного языка на путь в графе.

Для этого, во-первых, предъявим булевые представления для матриц смежности графа и автомата для регулярного языка. Затем, введем специальную

блочно–диагональную матрицу для синхронизации обхода в ширину по двум матрицам смежности. Далее, попробуем наивно реализовать обход в ширину, и посмотрим, почему наивная реализация может выдавать некорректный результат. После этого перейдем к реализации обхода в ширину более продвинутым методом, который решает проблему наивного подхода.

Пример 8.2.1. Возьмём следующий граф.



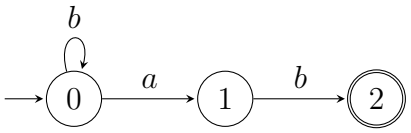
Его матрица смежности имеет следующий вид.

$$G_1 = \begin{pmatrix} . & \{a\} & . & \{b\} \\ . & . & \{b\} & . \\ \{a\} & . & . & . \\ \{b\} & . & . & . \end{pmatrix}$$

Её булева декомпозиция по каждому символу выглядит следующим образом.

$$G_{0\_a} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \quad G_{0\_b} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

Зададим ограничения с помощью регулярного выражения b^*ab, которое представляется автоматом из трех последовательных состояний.



Автомат может быть задан матрицей смежности (с дополнительной информацией о стартовых и финальных состояниях).

Для регулярного выражения b^*ab матрица смежности выглядит следующим образом (при этом нужно запомнить, что состояние 0 является начальным, 2 — конечным).

$$G_2 = \begin{pmatrix} \{b\} & \{a\} & . \\ . & . & \{b\} \\ . & . & . \end{pmatrix}$$

Нам будет необходима булева декомпозиция этой матрицы, и она выглядит следующим образом.

$$R_{0\_a} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad R_{0\_b} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Для синхронизации обхода составим набор блочно–диагональных матриц, каждая из которых — это прямая сумма двух матриц: $D_{0\_a} = R_{0\_a} \oplus G_{0\_a}$ и $D_{0\_a} = R_{0\_b} \oplus G_{0\_b}$.

$$D_{0\_a} = \left(\begin{array}{ccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) \quad D_{0\_b} = \left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right)$$

Пусть мы решаем частный случай задачи достижимости с несколькими стартовыми вершинами (multiple–source) — достижимость с одной стартовой вершиной (single–source).

Пусть единственной начальной вершиной в графе будет вершина 0.

Теперь создадим вектор $v = \boxed{1\ 0\ 0}\boxed{1\ 0\ 0\ 0}$, где в первой части стоит единица на месте начального состояния 0 в автомате. Во второй части содержится фронт обхода графа, на первом шаге это всегда множество стартовых вершин. В данном случае единица стоит на месте единственной стартовой вершины — 0.

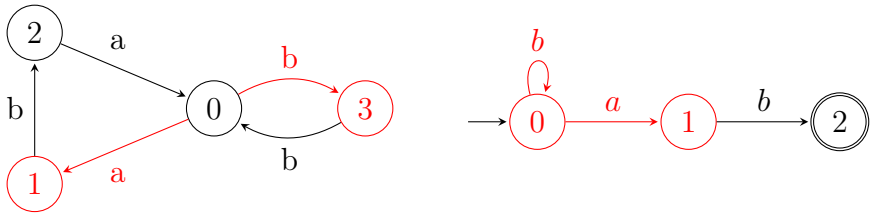
Совершим один шаг в обходе графа и получим новый фронт обхода графа.

$$a: \boxed{1\,0\,0}\boxed{1\,0\,0\,0} \times \left(\begin{array}{ccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) = \boxed{0\,1\,0}\boxed{0\,1\,0\,0}$$

$$b: \boxed{1\,0\,0}\boxed{1\,0\,0\,0} \times \left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right) = \boxed{1\,0\,0}\boxed{0\,0\,0\,1}$$

Сложим два полученных вектора, чтобы получить новый фронт обхода графа: $\boxed{0\,1\,0}\boxed{0\,1\,0\,0} + \boxed{1\,0\,0}\boxed{0\,0\,0\,1} = \boxed{1\,1\,0}\boxed{0\,1\,0\,1}$.

То есть в наш фронт $\boxed{0\,1\,0\,1}$ попали вершины 1 и 3 соотвественно. А именно, мы совершили следующие переходы в графе и автомате.

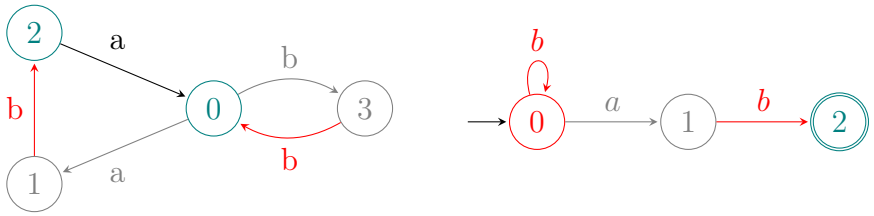


Совершим еще один шаг алгоритма. Теперь вектор v, на который мы умножаем матрицы, имеет следующий вид $\boxed{1\,1\,0}\boxed{0\,1\,0\,1}$.

$$a: \boxed{1\,1\,0}\boxed{0\,1\,0\,1} \times \left(\begin{array}{ccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) = \boxed{0\,1\,0}\boxed{0\,0\,0\,0}$$

$$b : \boxed{1\ 1\ 0}\boxed{0\ 1\ 0\ 1} \times \left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right) = \boxed{1\ 0\ 1}\boxed{1\ 0\ 1\ 0}$$

$\boxed{0\ 1\ 0}\boxed{0\ 0\ 0\ 0} + \boxed{1\ 0\ 1}\boxed{1\ 0\ 1\ 0} = \boxed{1\ 1\ 1}\boxed{1\ 0\ 1\ 0}$. То есть в наш фронт $\boxed{1\ 0\ 1\ 0}$ попали вершины 0 и 2 соотвественно. Мы совершили следующие переходы в графе и автомате.



При этом, можно заметить, что мы достигли конечной вершины в автомате. Последний элемент левой части результирующего вектора $\boxed{1\ 1\ 1}\boxed{1\ 0\ 1\ 0}$ отвечает за состояние 2, которое является конечным. А значит, обход необходимо остановить, и текущие вершины фронта обхода графа записать в ответ.

Таким образом, вершины графа 0 и 2 являются ответом. Однако вершина 0 — лишняя. Регулярное выражение b^*ab не подразумевает, что вершина 0 в графе может быть достигнута. Она могла бы быть достигнута по пустой строке в случае, если бы регулярное выражение имело вид b^* или по строке aba в случае, если бы регулярное выражение имело вид b^*aba.

Это произошло, потому что в векторе v должна кодироваться информация о паре — вершине графа и состоянии автомата. Достигнув вершины 0, мы оказались в конечном состоянии автомата, которое было получено с помощью другой вершины — вершины 2.

Эту проблему можно решить, закодировав информацию о каждой такой паре в несколько векторов v, и ограничив левую часть вектора v таким образом, чтобы в ней всегда была лишь одна единица.

Тогда мы получим, что вектор v вида $\boxed{1\ 0\ 0}\boxed{1\ 0\ 1\ 0}$ будет хранить информацию о парах (0, 0) и (0, 2), где первый элемент пары — состояние автомата, а второй — вершина графа.

Аналогично, вектор v вида $\boxed{0\ 1\ 0}\boxed{0\ 1\ 1\ 0}$ кодирует информацию о парах (1, 1) и (1, 2). Вектор v вида $\boxed{0\ 0\ 1}\boxed{0\ 0\ 1\ 1}$ кодирует информацию о парах (2, 2) и (2, 3).

Таким образом, мы будем понимать, в каком состоянии автомата мы находимся для каждой из вершин фронта обхода графа.

Рассмотрим, как это применяется в разработанном алгоритме, который представлен далее.

Предлагается "расклеить" v в матрицу M, состоящую из трех векторов, добавив два вектора $\boxed{0\ 1\ 0}\boxed{0\ 0\ 0\ 0}$ и $\boxed{0\ 0\ 1}\boxed{0\ 0\ 0\ 0}$. Во второй части этих векторов стоят нули, так как $\boxed{0\ 1\ 0}$ и $\boxed{0\ 0\ 1}$ кодируют состояния автомата 1 и 2, которые не являются начальными.

$$M = \begin{array}{|c|c|}\hline 1\ 0\ 0 & 1\ 0\ 0\ 0 \\ \hline 0\ 1\ 0 & 0\ 0\ 0\ 0 \\ \hline 0\ 0\ 1 & 0\ 0\ 0\ 0 \\ \hline \end{array}$$

И совершать обход тем же самым образом, но сохранив с помощью матрицы M дополнительную информацию о парах (состояние, вершина).

$$a: \begin{array}{|c|c|}\hline 1\ 0\ 0 & 1\ 0\ 0\ 0 \\ \hline 0\ 1\ 0 & 0\ 0\ 0\ 0 \\ \hline 0\ 0\ 1 & 0\ 0\ 0\ 0 \\ \hline \end{array} \times \left(\begin{array}{ccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) = \begin{array}{|c|c|}\hline 0\ 1\ 0 & 0\ 1\ 0\ 0 \\ \hline 0\ 0\ 0 & 0\ 0\ 0\ 0 \\ \hline 0\ 0\ 0 & 0\ 0\ 0\ 0 \\ \hline \end{array}$$

$$b: \begin{array}{|c|c|}\hline 1\ 0\ 0 & 1\ 0\ 0\ 0 \\ \hline 0\ 1\ 0 & 0\ 0\ 0\ 0 \\ \hline 0\ 0\ 1 & 0\ 0\ 0\ 0 \\ \hline \end{array} \times \left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right) = \begin{array}{|c|c|}\hline 1\ 0\ 0 & 0\ 0\ 0\ 1 \\ \hline 0\ 0\ 1 & 0\ 0\ 0\ 0 \\ \hline 0\ 0\ 0 & 0\ 0\ 0\ 0 \\ \hline \end{array}$$

Для того, чтобы левая часть матрицы M всегда оставалось единичной, нужно трансформировать в ней строчки особым образом. Для этого нужно

складывать только те вектора в правой части матрицы M, у которых в левой части единицы стоят на одинаковых позициях. После чего переставлять строчки в M так, чтобы левая часть матрицы M принимала единичный вид. Вектора с пустой левой частью нас при этом не интересуют.

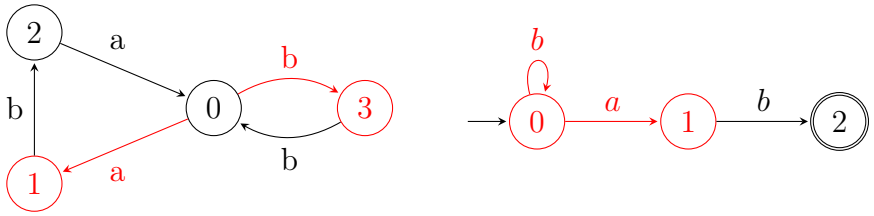
Тогда правая часть матрицы M будет кодировать текущий фронт обхода графа.

В нашем примере матрица M для следующего шага обхода выглядит следующим образом.

$$M = \left[\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right]$$

Видно, что во фронт обхода графа попали вершины 1 и 3. В вершину 1 мы попали в состоянии 1, в вершину 3 — в состоянии 0.

Совершаются следующие переходы в графе и автомате.



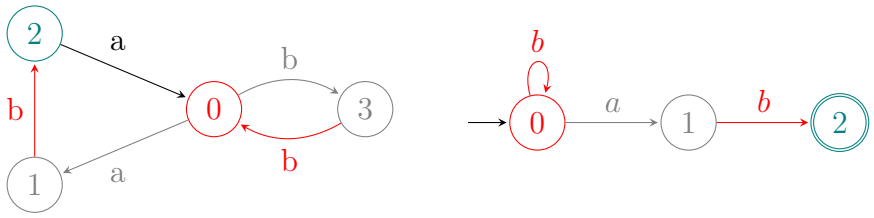
Сделаем еще один шаг алгоритма и придем к конечному состоянию в автомате.

$$a : \left[\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{array}\right] \times \left(\begin{array}{ccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right) = \left[\begin{array}{ccc|cccc} 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}\right]$$

$$b: \begin{array}{|c|c|}\hline 1\,0\,0 & 0\,0\,0\,1 \\ \hline 0\,1\,0 & 0\,1\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,0\,0 \\ \hline \end{array} \times \left(\begin{array}{ccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array}\right) = \begin{array}{|c|c|}\hline 1\,0\,0 & 1\,0\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,1\,0 \\ \hline 0\,0\,0 & 0\,0\,0\,0 \\ \hline \end{array}$$

$$M = \begin{array}{|c|c|}\hline 1\,0\,0 & 1\,0\,0\,0 \\ \hline 0\,1\,0 & 0\,0\,0\,0 \\ \hline 0\,0\,1 & 0\,0\,1\,0 \\ \hline \end{array}$$

Видно, что мы достигли вершины 2 графа в конечном состоянии 2 автомата. При этом вершина 0 графа так же достигнута, как и в наивном варианте алгоритма, но теперь известно, что это происходит в состоянии 0 автомата.



Таким образом, в ответ попадает вершина 2.

Перейдем к формальному описанию алгоритма.

Алгоритм принимает на вход граф $\mathcal{G}$, детерминированный конечный автомат $\mathcal{R}$, описывающий регулярную грамматику, и множество начальных вершин V_{src} графа.

Граф $\mathcal{G}$ и автомат $\mathcal{R}$ можно представить в виде булевых матриц смежности. Так, в виде словаря для каждой метки графа заводится булева матрица смежности, на месте (i, j) ячейки которой стоит 1, если i и j вершины графа соединены ребром данной метки. Такая же операция проводится для автомата грамматики $\mathcal{R}$.

Далее, мы оперируем с двумя словарями, где ключом является символ метки ребра графа или символ алфавита автомата, а значением — соответствующая им булевая матрица.

Для каждого символа из пересечения этих множеств строится матрица $\mathfrak{D}$, как прямая сумма булевых матриц. То есть, строится матрица $\mathfrak{D} = Bool_{\mathcal{R}_a} \bigoplus Bool_{\mathcal{G}_a}$, которая определяется как

$$\mathfrak{D} = \begin{bmatrix} Bool_{\mathcal{R}_a} & 0 \\ 0 & Bool_{\mathcal{G}_a} \end{bmatrix} \tag{8.1}$$

Где $\mathcal{R}_a$ и $\mathcal{G}_a$ матрицы смежности соответствующих символов автомата грамматики $\mathcal{R}$ и графа $\mathcal{G}$ для символа $a \in A_{\mathcal{R}} \cap A_{\mathcal{G}}$, $A_{\mathcal{R}} \cap A_{\mathcal{G}}$ — пересечение алфавитов. Такая конструкция позволяет синхронизировать алгоритм обхода в ширину одновременно для графа и грамматики.

Далее вводится матрица M, хранящая информацию о фронте обхода графа. Она нужна для выделения множества пройденных вершин и не допускает зацикливание алгоритма.

$$M^{k\times(k+n)} = \begin{bmatrix} Id_k & Matrix_{k\times n} \end{bmatrix} \tag{8.2}$$

Где Id_k — единичная матрица размера k, k — количество вершин в автомате $\mathcal{R}$, $Matrix_{k\times n}$ — матрица, хранящая в себе маску пройденных вершин в автомате графа, n — количество вершин в графе $\mathcal{G}$.

8.2.1 Выходные данные

На выходе строится множество $\mathcal{P}$ пар вершин (v, w) графа $\mathcal{G}$ таких, что вершина w достижима из множества начальных вершин, при этом $v \in V_{src}$, $w \notin V_{src}$. Это множество представляется в виде матрицы размера $|V| \times |V|$, где (i, j) ячейка содержит 1, если пара вершин с индексами $(i, j) \in \mathcal{P}$.

8.2.2 Процесс обхода графа

Алгоритм обхода заключается в последовательном умножении матрицы M текущего фронта на матрицу $\mathfrak{D}$. В результате чего, находится матрица M' содержащая информацию о вершинах, достижимых на следующем шаге. Далее, с помощью операций перестановки и сложения векторов M' преобразуется к виду матрицы M и присваивается ей. Итерации продолжаются пока M' содержит новые вершины, не содержащиеся в M. На листинге 3 представлен этот алгоритм.

В алгоритме 3, в 10 строке происходит трансформация строчек в матрице M'. Это делается для того, чтобы представить полученную во время обхода матрицу M', содержащую новый фронт, в виде матрицы M. Для этого

Algorithm 3 Алгоритм достижимости в графе с регулярными ограничениями на основе поиска в ширину, выраженный с помощью операций матричного умножения

```
1: procedure BFSBasedRPQ(R = ⟨Q, Σ, P, F, q⟩, G = ⟨V, E, L⟩, V_src)
2:     P ← Матрица смежности графа
3:     𝔇 ← Bool_R ⨁ Bool_G                          ▷ Построение матриц 𝔇
4:     M ← CreateMasks(|Q|, |V|)                    ▷ Построение матрицы M
5:     M' ← SetStartVerts(M, V_src)                 ▷ Заполнение нач. вершин
6:     while Матрица M меняется do
7:         M ← M'⟨¬M⟩                   ▷ Применение комплементарной маски
8:         for all a ∈ (Σ ∩ L) do
9:             M' ← M any.pair 𝔇                ▷ Матр. умножение в полукольце
10:            M' ← TransformRows(M')           ▷ Приведение M' к виду M
11:        Matrix ← extractRightSubMatrix(M')
12:        V ← Matrix.reduceVector()                ▷ Сложение по столбцам
13:        for k ∈ 0 . . . |V_src| − 1 do
14:            W ← P.getRow(k)
15:            P.setRow(k, V + W)
16:    return P
17: end procedure
```

требуется так переставить строчки M', чтобы она содержала корректные по своему определению значения. То есть, имела единицы на главной диагонали, а все остальные значения в первых k столбцах были нулями. Подробнее эта процедура описана в листинге 4.

Algorithm 4 Алгоритм трансформации строчек

```
1: procedure TransformRows(M)
2:     T ← extractLeftSubMatrix(M)
3:     Ix, Iy ← итераторы по индексам ненулевых элементов T
4:     for i ∈ 0 . . . |Iy| do
5:         R ← M.getRow(Ix[i])
6:         M'.setRow(Iy[i], R + M'.getRow(Iy[i]))
7: end procedure
```

Algorithm 5 Модификация алгоритма для поиска конкретной исходной вершины

procedure BFSBasedRPQ($\mathcal{R} = \langle Q, \Sigma, P, F, q\rangle, \mathcal{G} = \langle V, E, L\rangle, V_{src}$)
  $\mathcal{P} \leftarrow$ Матрица смежности графа
  $\mathfrak{D} \leftarrow Bool_{\mathcal{R}} \bigoplus Bool_{\mathcal{G}}$
  $\mathfrak{M} \leftarrow CreateMasks(|Q|, |V|)$
  $\mathfrak{M}' \leftarrow SetStartVerts(\mathfrak{M}, V_{src})$
  while Матрица $\mathfrak{M}$ меняется **do**
    $\mathfrak{M} \leftarrow \mathfrak{M}'\langle\neg\mathfrak{M}\rangle$
    for all $a \in (\Sigma \cap L)$ **do**
      $\mathfrak{M}' \leftarrow \mathfrak{M}$ any.pair $\mathfrak{D}$
      for all $M \in \mathfrak{M}'$ **do**
        $M \leftarrow TransformRows(M)$
    for all $M_k \in \mathfrak{M}'$ **do**
      $Matrix \leftarrow extractSubMatrix(M)$
      $V \leftarrow Matrix.reduceVector()$
      $W \leftarrow \mathcal{P}.getRow(k)$
      $\mathcal{P}.setRow(k, V + W)$
  return $\mathcal{P}$
end procedure

8.2.3 Модификации алгоритма

Рассмотрим V_{src} — множество начальных вершин, состоящее из r элементов. Для каждой начальной вершины $v^i_{src} \in V_{src}$ отметим соответствующие индексы в матрице M единицами, получив матрицу $M(v^i_{src})$, и построим матрицу $\mathfrak{M}$ следующим образом.

$$\mathfrak{M}^{(k*r)\times(k+n)} = \begin{bmatrix} M(v^1_{src}) \\ M(v^2_{src}) \\ M(\dots) \\ M(v^r_{src}) \end{bmatrix} \tag{8.3}$$

Матрица $\mathfrak{M}$ собирается из множества матриц $M(v^i_{src})$ и позволяет хранить информацию о том, из какой начальной вершины достигаются новые вершины во время обхода.

В листинге 5 представлен модифицированный алгоритм. Основное его отличие заключается в том, что для каждой достижимой вершины находится конкретная исходная вершина, из которой начинался обход.

Таким образом, алгоритмы 3 и 5 решают сформулированные в пункте **??** задачи достижимости.

Глава 9

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.

9.1 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 грамматике [39].

Для его применения необходимо, чтобы подаваемая на вход грамматика находилась в Нормальной Форме Хомского (НФХ) 5.3. 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. Из нетерминала A выводится цепочка ω при помощи правила $A \to a$ тогда и только тогда, когда $a = \omega$:

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

2. Из нетерминала A выводится цепочка ω при помощи правила $A \to BC$ 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 $\omega = \omega_1\omega_2$:

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

Переформулируем эти утверждения 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 9.1.1. Рассмотрим пример работы алгоритма CYK на грамматике правильных скобочных последовательностей в Нормальной Форме Хомского.

$$\begin{array}{lll} S \to AS_2 \mid \varepsilon & S_2 \to b \mid BS_1 \mid S_1S_3 & A \to a \\ S_1 \to AS_2 & S_3 \to b \mid BS_1 & B \to b \end{array}$$

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 9.1.2. Теперь выполним алгоритм на цепочке $\omega = abaa$.

Шаг 1. Инициализируем таблицу:

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

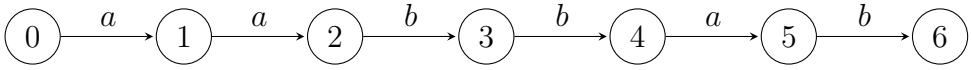
Шаг 2. Заполняем следующую диагональ:

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

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.

9.2 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 9.1):



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 9.1.1 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 (5.3.3).

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 грамматике есть правила вида

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

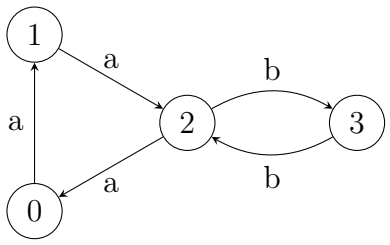
Затем используем матрицу смежности получившегося графа (обозначим ее M) в качестве начального значения. Дальнейший ход алгоритма можно описать псевдокодом, представленным в листинге 6.

Algorithm 6 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 грамматики.

Пример 9.2.1. Рассмотрим работу алгоритма на графе

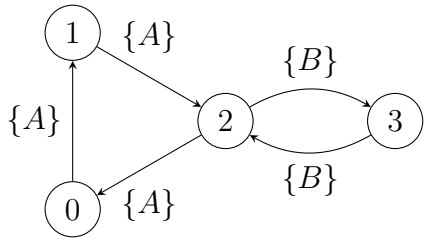


и грамматике:

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

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 [35], также она реализована в работе [82].

Псевдокод алгоритма Хеллингса представлен в листинге 7.

Algorithm 7 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 9.2.2. 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 грамматике.

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.

Глава 10

Контекстно-свободная достижимость через произведение матриц

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

10.1 Алгоритм контекстно-свободной достижимости через произведение матриц

В главе 9.2 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 [72], 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 грамматик за субкубическое время, его обобщение до задачи КС достижимости в произвольных графах с сохранением асимптотики является нетривиальной задачей [79]. В настоящее время алгоритм с субкубической сложностью получен только для частного случая — языка Дика с одним типом скобок — Филипом Брэдфордом [16].

В случае с линейным входом, отдельного внимания заслуживает работа Лиллиан Ли (Lillian Lee) [46], где она показывает, что задача перемножения матриц сводима к синтаксическому анализу линейного входа. Аналогичных результатов для графов на текущий момент не известно.

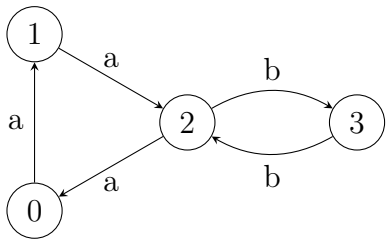
Поэтому рассмотрим более простую идею, изложенную в статье Рустама Азимова [6]: будем строить транзитивное замыкание графа через наивное (не через возведение в квадрат) умножение матриц.

Пусть $\mathcal{G} = (V, E)$ — входной граф и $G = (N, \Sigma, P)$ — входная грамматика. Тогда алгоритм может быть сформулирован, как представлено в листинге 8.

Algorithm 8 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:     for all i ∈ 0 ... n − 1 do        ▷ Добавление петель для нетерминалов,
   порождающих пустую строку
9:         T_{i,i} ← T_{i,i} ∪ {A ∈ N | A → ε}
10:    while матрица T меняется do
11:        T ← T ∪ (T × T)                  ▷ Вычисление транзитивного замыкания
12:    return T
```

Пример 10.1.1 (Пример работы)**.** Пусть есть граф $\mathcal{G}$:



и грамматика G:

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

Пусть T_i — матрица, полученная из T после применения цикла, описанного в строках **8-9** алгоритма 8, 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 8 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.

10.1.1 Расширение алгоритма для конъюнктивных грамматик

Матричный алгоритм для конъюнктивных грамматик отличается от алгоритма 8 для контекстно-свободных грамматик только операцией умножения матриц, в остальном алгоритм остается без изменений. Определим операцию умножения матриц.

Определение 10.1.1. Пусть M_1 и M_2 матрицы размера n. Определим операцию $\circ$ следующим 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

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

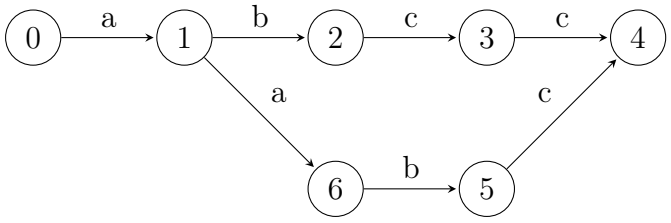
Важно заметить, что алгоритм для конъюнктивных грамматик, в отличие от алгоритма для контекстно-свободных грамматик, дает лишь верхнюю оценку ответа. То есть все нетерминалы, которые должны быть в ячейках матрицы результата, содержатся там, но вместе с ними содержатся и лишние нетерминалы. Рассмотрим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

Пример 10.1.2. 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 грамматик такую проверку реализовать нельзя. Поэтому для классической семантики достижимости с ограничениями в терминах конъюнктивных грамматик результат работы алгоритма является оценкой сверху.

Существует альтернативная семантика, когда мы трактуем конъюнкцию в правой части правил как конъюнкцию в Datalog (подробнее о Datalog в

параграфе **??**). 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 [7]. Стоит также отметить, что обобщения данного алгоритма для булевых грамматик не существует, хотя и существует частное решение для случая, когда граф не содержит циклов (является DAG-ом), предложенное Екатериной Шеметовой [65].

10.2 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 [48]. Это позволяет с минимальными затратами 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 алгоритмов 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 содержится этот нетерминал. Тогда перемножение пары таких матриц, соответствующих нетерминалам A и B, соответствует построению путей, выводимых из нетерминалов, для которых есть правила с правой частью вида AB.

Пример 10.2.1. 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 9. Такой взгляд на алгоритм позволяет использовать для его реализации существующие высокопроизводительные библиотеки для работы с булевыми матрицами (например M4RI[1] [2]) или библиотеки для линейной алгебры (например CUSP [23]).

Listing 9 Context-free path querying 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

С другой стороны, для запросов, выразимых в терминах грамматик 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:

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

[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.

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.

Пример 10.2.2. 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 библиотеки линейной 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 SuiteSparse[3] [25].

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

[3]SuiteSparse — 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.

Также стоит заметить, что при работе с реальными графами матрицы, как правило, оказываются разреженными, а значит необходимо использовать соответствующие представления матриц (CRS, покоординатное, Quad Tree [29]) и библиотеки, работающие с таким представлениями.

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

Глава 11

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.

11.1 Тензорное произведение

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

Определение 11.1.1. 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 естественным образом, если считать, что пары вершин, задающие ребро, упорядочены, поэтому формальное определение отличаться не будет.

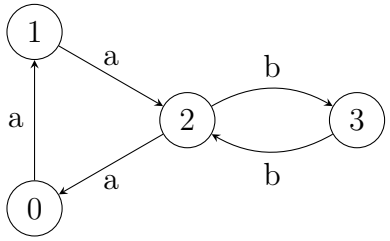
Осталось добавить метки к рёбрам. Это приведёт к логичному усилению требования к существованию ребра: метки рёбер в исходных графах должны совпадать. Таким образом, мы получаем следующее определение тензорного произведения ориентированных графов с метками на рёбрах.

Определение 11.1.2. 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 11.1.1. Рассмотрим пример. В качестве одного из графов возьмём рекурсивный автомат, построенный ранее (изображение 5.1). Его матрица смежности выглядит следующим образом.

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



(11.1)

Второй граф представлен на изображении 11.1. Его матрица смежности имеет следующий вид.

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

Теперь вычислим $M_1 \otimes M_2$.

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

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

11.2 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 замечании 1.7.1, существует решение этой проблемы.

Будем рассматривать два конечных автомата: одни получен 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 грамматике 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 10.

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 при помощи функции $getCoordinates(i, j)$.

Представленный алгоритм не требует преобразования грамматики в ОНФХ, более того, рекурсивный автомат может быть минимизирован. Однако, результатом тензорного рпоизведения является матрица существенно бóльшего размера, чем в алгоритме, основанном на матричном произведении. 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 грамматике мы не можем определить, задаёт ли она регулярный язык, то при добавлении в язык запросов возможности 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 запросов, что может приводить к существенным накладным расходам при выполнении ре-

Listing 10 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
```

гулярного запроса. Данный же алгоритм не выполнит лишних действий, так как сразу выполнит классическое пересечение двух автоматов и получит результат.

11.3 Примеры

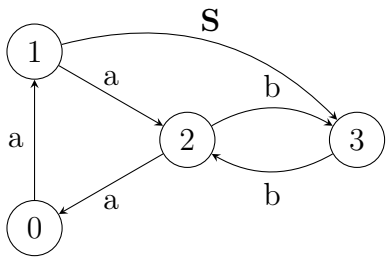
Рассмотрим подробно ряд примеров работы описанного алгоритма. Будем для каждой итерации внешнего цикла выписывать результаты основных операций: тензорного произведения, транзитивного замыкания, обновления матрицы смежности входного графа. Новые, по сравнению с предыдущим состоянием, элементы матриц будем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

Пример 11.3.1. Теоретически худший случай. Такой же как и для матричного.

Итерация 1 (конец). Начало в разделе 11.1, где мы вычислили тензорное произведение матриц смежности. Теперь нам осталось только вычислить транзитивное замыкание полученной матрицы:

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

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:

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

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

Итерация 2. Вычисляем тензорное произведение матриц смежности.

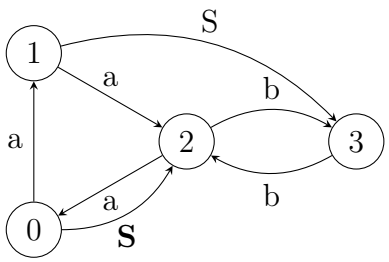
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . \\ . & . & [S] & [b] \\ . & . & . & [b] \\ . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & [a] & . & . \\ . & . & [a] & [S] \\ [a] & . & . & [b] \\ . & . & [b] & . \end{pmatrix} =$$

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

Вычисляем транзитивное замыкание полученной матрицы:

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

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.

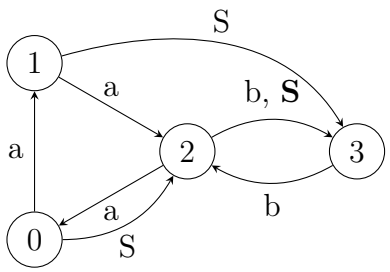
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . \\ . & . & [S] & [b] \\ . & . & . & [b] \\ . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & [a] & [S] & . \\ . & . & [a] & [S] \\ [a] & . & . & [b] \\ . & . & [b] & . \end{pmatrix} =$$

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

Затем вычисляем транзитивное замыкание:

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

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.

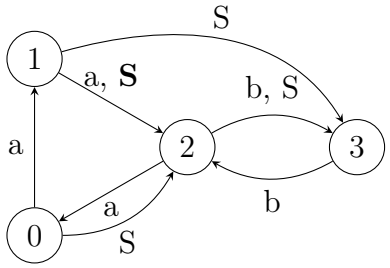
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . \\ . & . & [S] & [b] \\ . & . & . & [b] \\ . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & [a] & [S] & . \\ . & . & [a] & [S] \\ [a] & . & . & [b, S] \\ . & . & [b] & . \end{pmatrix} =$$

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

Затем транзитивное замыкание:

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

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

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

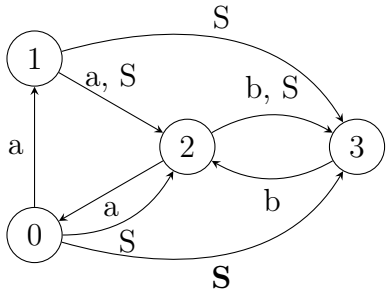
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} \cdot & [a] & \cdot & \cdot \\ \cdot & \cdot & [S] & [b] \\ \cdot & \cdot & \cdot & [b] \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \otimes \begin{pmatrix} \cdot & [a] & [S] & \cdot \\ \cdot & \cdot & [a,S] & [S] \\ [a] & \cdot & \cdot & [b,S] \\ \cdot & \cdot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

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

Затем вычисляем транзитивное замыкание:

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

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.

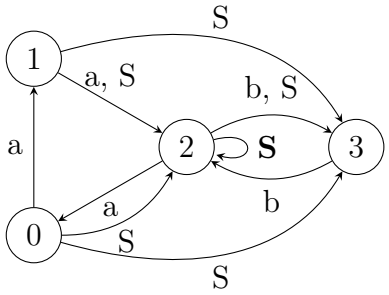
$$M_3 = M_1 \otimes M_2 = \begin{pmatrix} . & [a] & . & . \\ . & . & [S] & [b] \\ . & . & . & [b] \\ . & . & . & . \end{pmatrix} \otimes \begin{pmatrix} . & [a] & [S] & [S] \\ . & . & [a,S] & [S] \\ [a] & . & . & [b,S] \\ . & . & [b] & . \end{pmatrix} =$$

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

Транзитивное замыкание:

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

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 9.2.1).

Пример 11.3.2. 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 грамматика классическая:

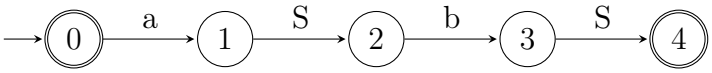
$$G_1 = \langle \{a,\ b\}, \{S\}, \{S \to a\ S\ b\ S \mid \varepsilon\} \rangle$$

Во второй грамматике мы будем использовать конструкции регулярных выражений в правой части правил. То есть вторая грамматика находится в EBNF (Расширенная форма Бэкуса-Наура [37, 77]).

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

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

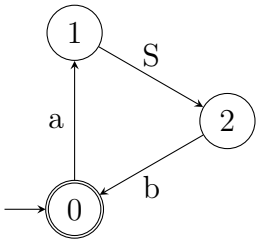
Рекурсивный автомат N_1 для грамматики G_1:



Матрица смежности N_1:

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

Рекурсивный автомат N_2 для грамматики G_2:

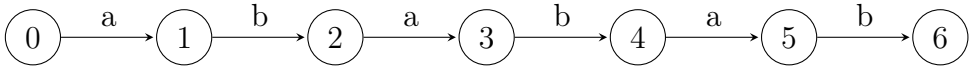


Матрица смежности N_2:

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

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:

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

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

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

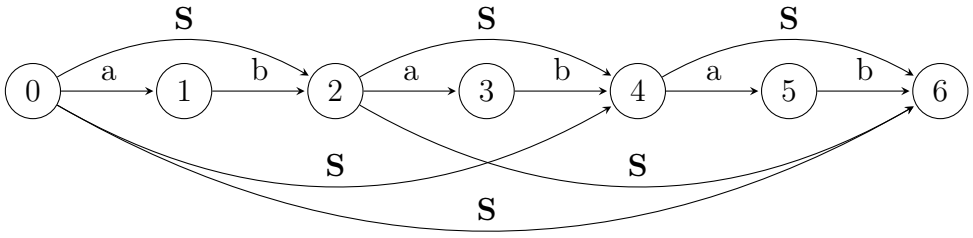
$$= \left(\begin{array}{ccccccc|ccccccc|ccccccc}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & [a] & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \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 & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\cdot & \cdot & \cdot & \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 & \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 & \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
\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 & \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
\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:

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

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.

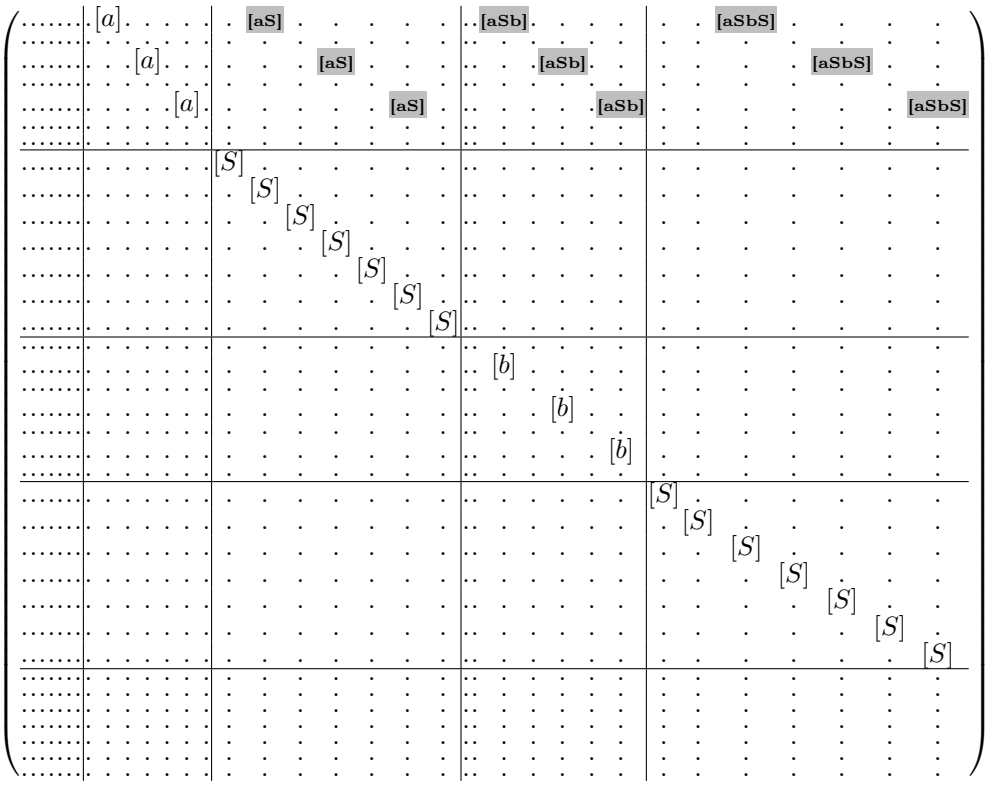
Теперь запустим алгоритм на второй грамматике и том же входе.

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

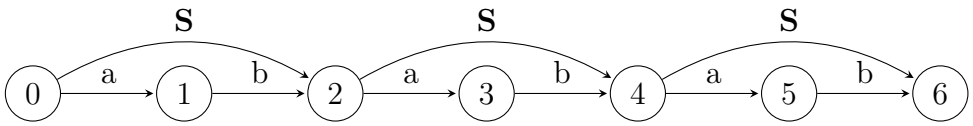
$$= \left(\begin{array}{c|c|c|c} \cdots & [a] \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots [a] \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots [a] \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots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

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:

$tc(M_3) =$ 

Обновлённый граф:



Его матрица смежности:

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

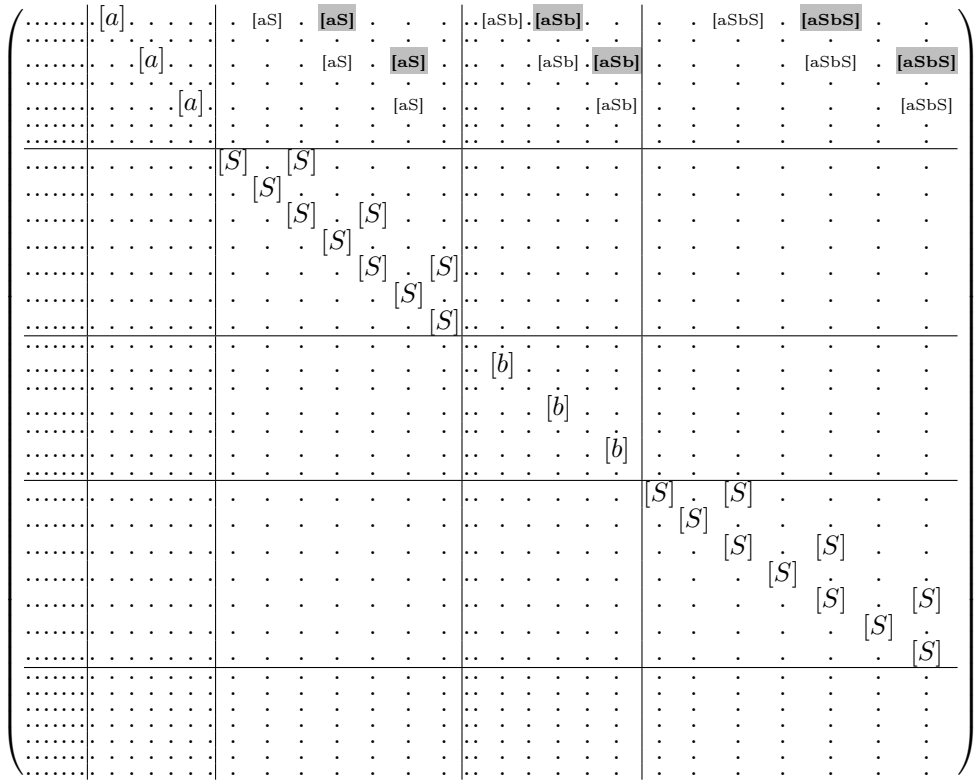
Потребуется ещё одна итерация.

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

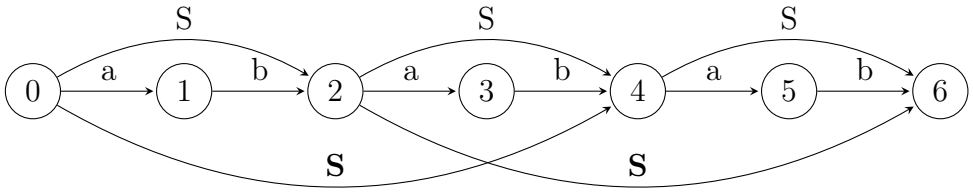
$$= \left(\begin{array}{c|c|c|c} \cdots & [a] \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots [a] \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots [a] \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \cdots & \cdots & \cdots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

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:

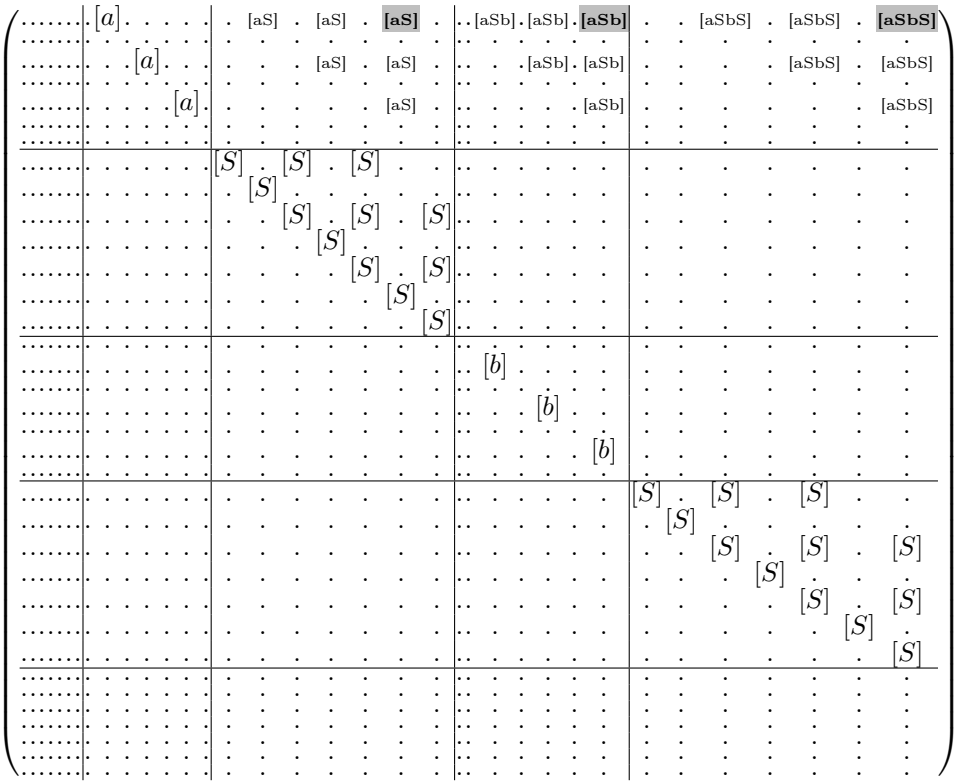
$$M_2 = \begin{pmatrix} [S] & [a] & [S] & . & \mathbf{[S]} & . & \\ . & [S] & [b] & . & . & . & \\ . & . & [S] & [a] & [S] & . & \mathbf{[S]} \\ . & . & . & [S] & [b] & . & . \\ . & . & . & . & [S] & [a] & [S] \\ . & . & . & . & . & [S] & [b] \\ . & . & . & . & . & . & [S] \end{pmatrix}$$

И, наконец, последняя содержательная итерация.

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

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

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.

11.4 Особенности реализации

Как и алгоритмы, представленные в разделе 10, 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.

Глава 12

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 ограничениям?

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 контекстно-свободный язык можно описать контекстно-свободной грамматикой, которая конечна. Это и есть решение нашего вопроса. Осталось только научиться строить такую грамматику.

Прежде, чем двинуться дальше, рекомендуется вспомнить всё, что касается деревьев вывода 5.1.

12.1 Лес разбора как представление контекстно-свободной грамматики

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

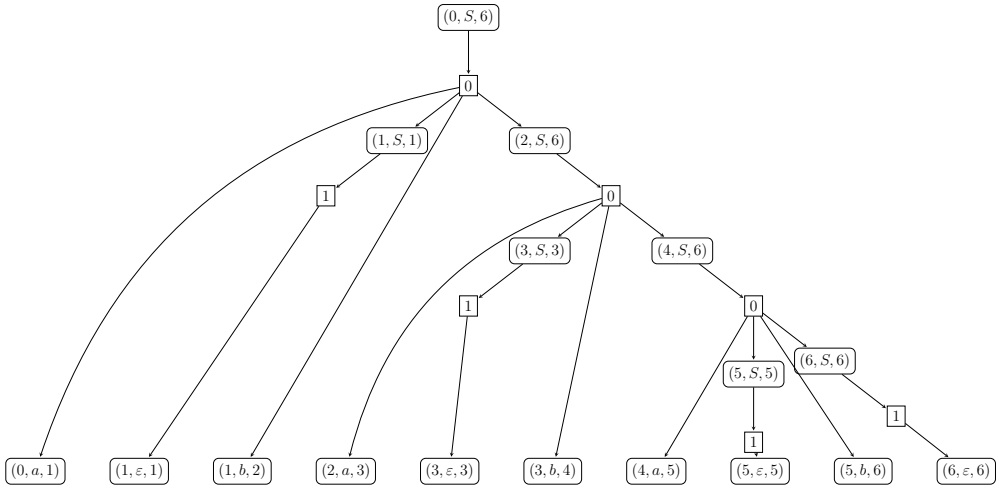
Во-первых, заметим, что в дереве вывода каждый узел соответствует выводу какой-то подстроки с известными позициями начала и конца. Давайте будем сохранять эту информацию в узлах дерева. Таким образом, метка любого узла это тройка вида (i, q, j), где i — координата начала подстроки, со-

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 12.1.1. 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 12.1.2. Сжатие леса вывода. Построим несколько деревьев вывода цепочки ${}_0a_1b_2a_3b_4a_5b_6$ в грамматике

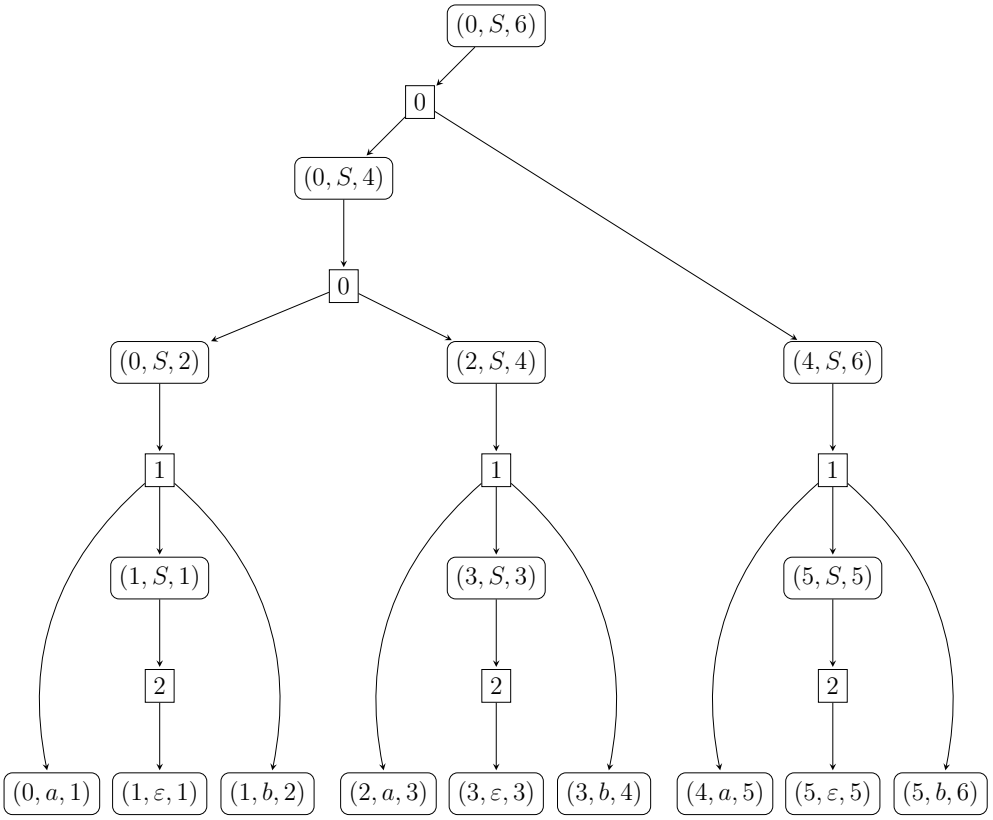
$$G_1 = \langle \{a, b\}, \{S\}, S, \{ \\ (0) S \to SS, \\ (1) S \to a\ S\ b, \\ (2) S \to \varepsilon \\ \}\rangle$$

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

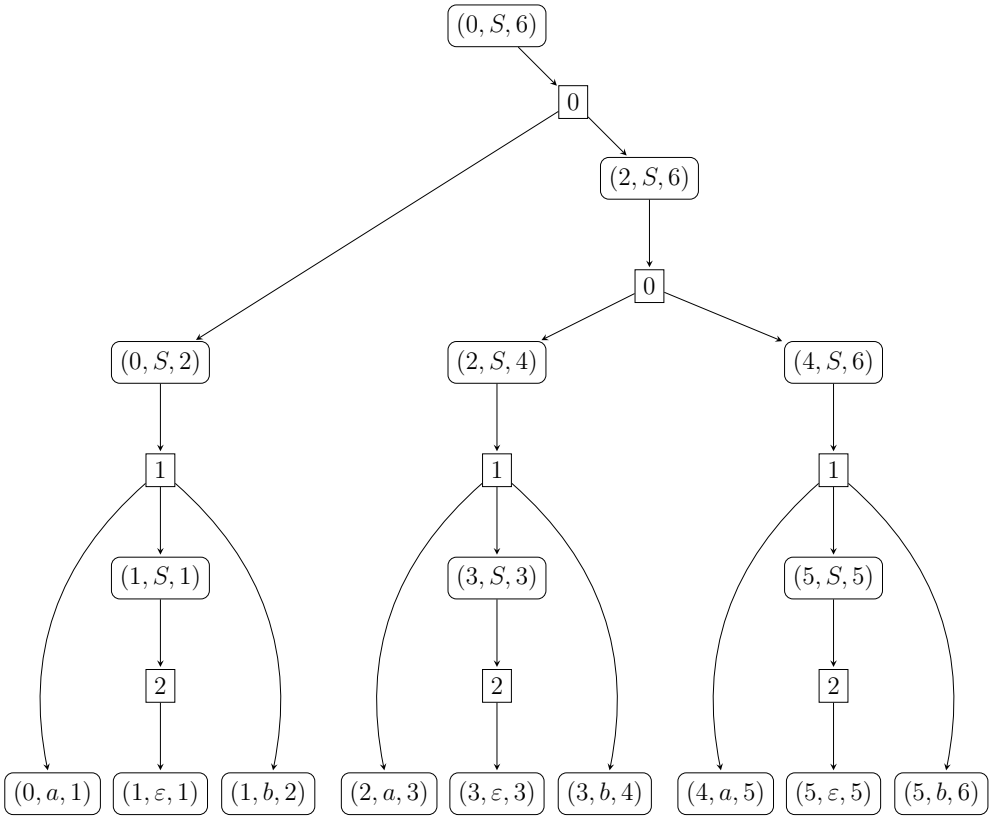
$$\mathbf{S} \xrightarrow{0} \mathbf{S}S \xrightarrow{0} \mathbf{S}SS \xrightarrow{1} a\mathbf{S}bSS \xrightarrow{2} ab\mathbf{S}S \xrightarrow{1} aba\mathbf{S}bS \xrightarrow{2} abab\mathbf{S} \xrightarrow{1} ababa\mathbf{S}b \xrightarrow{2} ababab$$

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

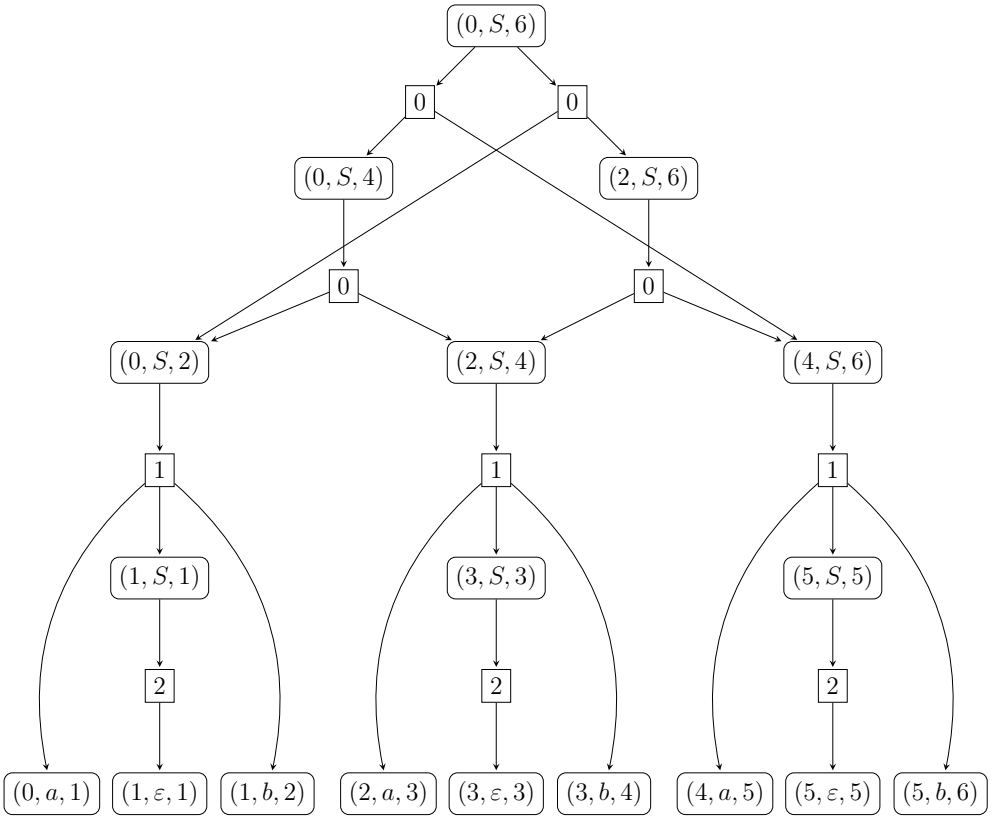
Сначала рассмотрим первый вариант (применили переписывание по продукции 0). Все остальные шаги вывода деретерминированы и в результате мы получим следующее дерево разбора:



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 получим 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 [54]. В дальнейшем она нашла широкое применение в обобщённом (generalized) синтаксическом анализе и получила серьёзное развитие. В частности, наш вариант, хоть и позволяет избежать экспоненциального разрастания леса разбора, всё же не является оптимальным. Оптимальное асимптотическое поведение достигается при использовании бинаризованного SPPF [14] — 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 [59], бинаризованная верся SPPF в BRNGLR [62] и GLL [60, 1][1].

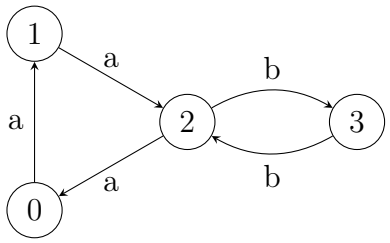
В действительности SPPF может содержать в себе циклы. Для линейного входа их можно получить, когда есть возможность выводить по грамматике бесконечные эпсилон-цепочки. Циклы будут вырожденными, но они будут.

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



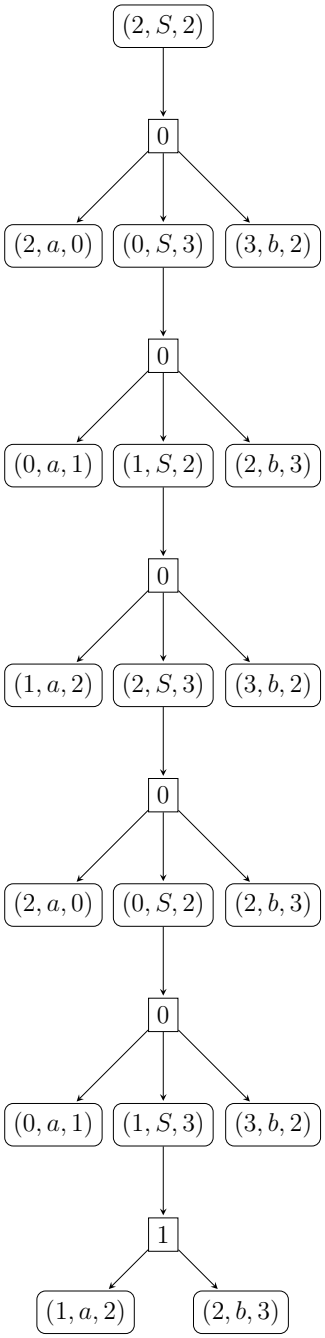
Дана грамматика

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

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

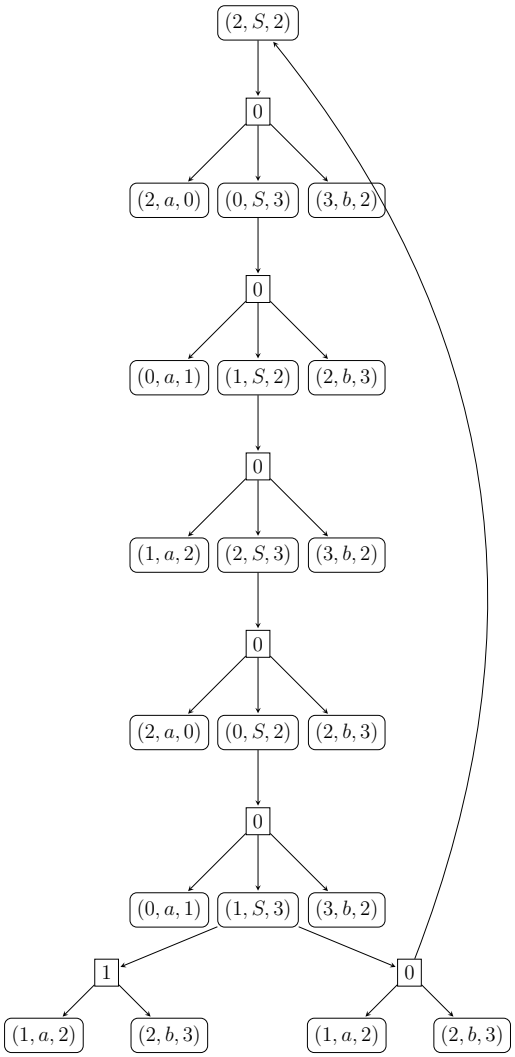
$$2 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{a} 2 \xrightarrow{b} 3 \xrightarrow{b} 2 \xrightarrow{b} 3 \xrightarrow{b} 2 \xrightarrow{b} 3 \xrightarrow{b} 2.$$

Построим дерево его вывода.



Мы построили дерево вывода для одного пути из вершины 2 в неё же. Но можно заметить, что таких путей бесконечно много: мы можем бесконечное число раз повторять 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 12.1.1. SPPF построенный для данной контекстно-свободной грамматики 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 12.1.4. 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:

$$_{\mathbf{2}}\boldsymbol{S}_{\mathbf{2}} \overset{(0)}{\Longrightarrow}$$

$$_2a_0\ _{\mathbf{0}}\boldsymbol{S}_{\mathbf{3}}\ _3b_2 \overset{(1)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _{\mathbf{1}}\boldsymbol{S}_{\mathbf{2}}\ _2b_3\ _3b_2 \overset{(2)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _{\mathbf{2}}\boldsymbol{S}_{\mathbf{3}}\ _3b_2\ _2b_3\ _3b_2 \overset{(3)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _2a_0\ _{\mathbf{0}}\boldsymbol{S}_{\mathbf{2}}\ _2b_3\ _3b_2\ _2b_3\ _3b_2 \overset{(4)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _{\mathbf{1}}\boldsymbol{S}_{\mathbf{3}}\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2 \overset{(5)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _{\mathbf{2}}\boldsymbol{S}_{\mathbf{2}}\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2 \overset{(0)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2a_0\ _{\mathbf{0}}\boldsymbol{S}_{\mathbf{3}}\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2 \overset{(1)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _{\mathbf{1}}\boldsymbol{S}_{\mathbf{2}}\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2 \overset{(2)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _{\mathbf{2}}\boldsymbol{S}_{\mathbf{3}}\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2 \overset{(3)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2a_0\ _{\mathbf{0}}\boldsymbol{S}_{\mathbf{2}}\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2 \overset{(4)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _{\mathbf{1}}\boldsymbol{S}_{\mathbf{3}}\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2 \overset{(6)}{\Longrightarrow}$$

$$_2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2a_0\ _0a_1\ _1a_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2\ _2b_3\ _3b_2$$

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

Глава 13

Алгоритм на основе нисходящего анализа

В данном разделе мы рассмотрим семейство алгоритмов нисходящего синтаксического (рекурсивный спуск, LL, GLL [60, 1]) и их обобщение для задачи поиска путей с контекстно-свободными ограничениями.

13.1 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 распознавателя 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 такого подхода есть два ограничения:

1. Не работает с леворекурсивными грамматиками, то есть грамматиками, вывод в которых 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 13.1.1. Построим функцию рекурсивного спуска для продукции $S \to aSbS \mid \varepsilon$.

Listing 11 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.

13.2 LL(k)-алгоритм синтаксического анализа

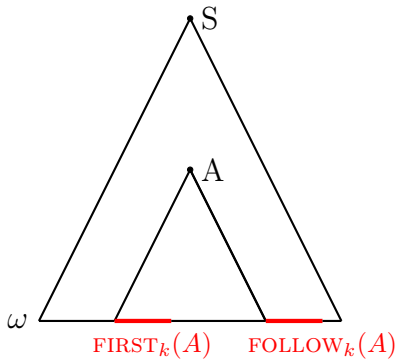
LL(k) — алгоритм синтаксического анализа — нисходящий анализ без отката, но с предпросмотром. Решение о том, какую продукцию применять, принимается на основании k следующих за текущим символом. Временная сложность алгоритма $O(n)$, где n — длина слова.

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 13.2.1. Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика. Множество FIRST_k определено для сентенциальной 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 13.2.2. Пусть $G = \langle N, \Sigma, P, S \rangle$ — КС-грамматика. Множество FOLLOW_k определено для сентенциальной 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 правила $A \to \alpha \mid \beta$

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 13.2.1. 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 13.2.2. 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 13.2.3. 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 стек пуст, а позиция во входной строке соответствует её началу. На первом шаге в стек добавляются последовательно сперва символ конца 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 текущая 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.

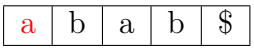
- Если текущая вершина стека — нетерминал N_i и текущий входной символ t_j, то ищем в управляющей таблице ячейку с координатами (N_i, t_j) и записываем на стек содержимое этой ячейки.

Пример 13.2.3. 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:

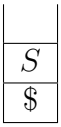
1. Начало работы.

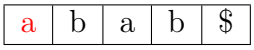
   Стек: 

   входное слово: 

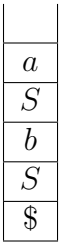
   Финальный символ лежит на стеке, а указатель указывает на первый символ слова.

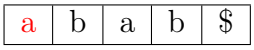
2. кладем стартовый символ на стек

   Стек: 

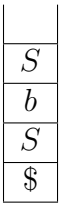
   входное слово: 

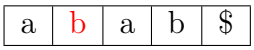
3. Ищем ячейку с координатами (S, a), применяем продукцию из ячейки.

   Стек: 

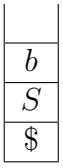
   входное слово: 

4. Снимаем терминал a со стека и двигаем указатель.

   Стек: 

   входное слово: 

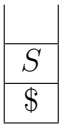
5. Ищем ячейку с координатами (S, b), применяем продукцию из ячейки.

Стек: 

входное слово:

a	b	a	b	$

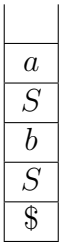
6. Снимаем терминал b со стека и двигаем указатель.

Стек: 

входное слово:

a	b	a	b	$

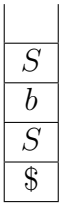
7. Ищем ячейку с координатами (S, a), применяем продукцию из ячейки.

Стек: 

входное слово:

a	b	a	b	$

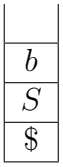
8. Снимаем терминал a со стека и двигаем указатель.

Стек: 

входное слово:

a	b	a	b	$

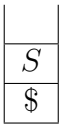
9. Ищем ячейку с координатами (S, b), применяем продукцию из ячейки.

Стек: 

входное слово:

a	b	a	b	$

10. Снимаем терминал b со стека и двигаем указатель.

Стек: 

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 терминал (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 алгоритмов всё так же не работает с леворекурсивными грамматиками и с неоднозначными грамматиками.

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

13.3 Алгоритм Generalized LL

Можно построить анализатор, работающий с произвольными КС-грамматиками. Generalized LL (GLL) [60, 1]

Принцип работы остается абсолютно таким же, как 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 13.3.1. 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 [71] 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 13.3.2. 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 [1], 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 будем 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 добавляем во 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 одного 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, таким образом терминируя текущую ветвь разбора.

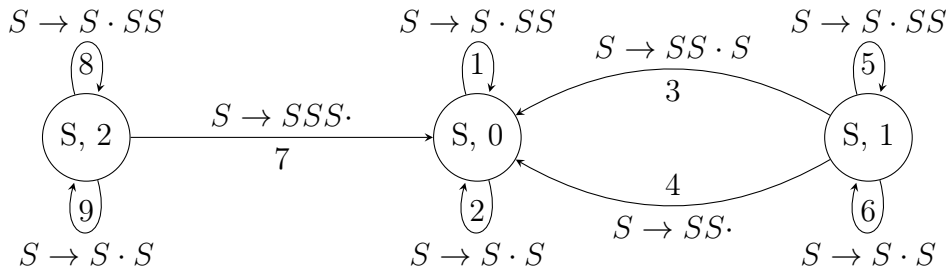
- $X ::= \alpha \cdot A\beta$, т.е. указатель смотрит на нетерминал. Нам нужен GSS узел v вида (A, i) и ребро $(u, X ::= \alpha A \cdot \beta, v)$ (ребро из u в v с пометкой $X ::= \alpha A \cdot \beta$). Если такой узел и ребро уже существуют в нашем GSS, берем их, иначе — создаём. Далее в R добавляем 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 13.3.1. 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:

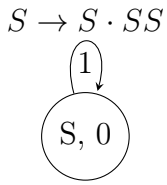
$S \to SSS \quad (0)$
$S \to SS \quad (1)$
$S \to a \quad (2)$

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

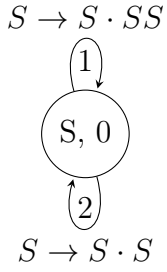
Разумеется, что конкретный порядок исполнения алгоритма будет зависеть, например, от используемой в качестве рабочего множества R структуры данных и от порядка обработки правил из ячейки управляющей таблицы. Рассмотрим лишь один из возможных вариантов:

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.

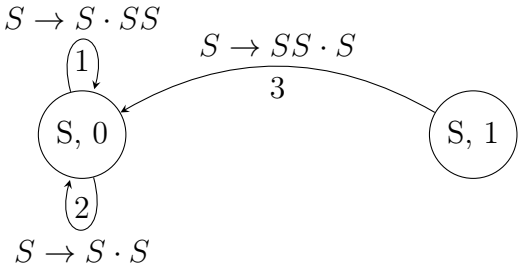


3. При обработке $(S \to \cdot SS, s_0, 0)$ образовывается петля 2, а в остальном аналогично шагу 2.

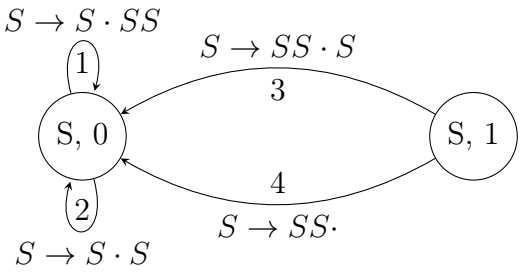


4. При обработке $(S \to \cdot a, s_0, 0)$ мы распознаем терминал a на позиции 0 и, возвращаясь по петлям 1 и 2, добавляем дескрипторы $(S \to S \cdot SS, s_0, 1), (S \to S \cdot S, s_0, 1)$.

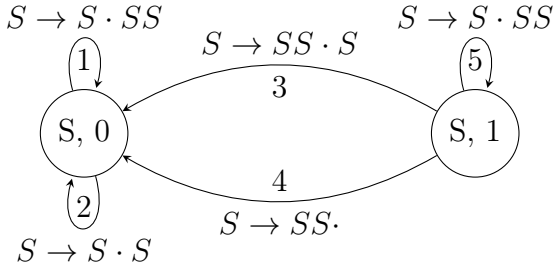
5. При обработке $(S \to S \cdot SS, s_0, 1)$ образовываем узел $s_1 = (S, 1)$ с исходящим ребром 3 и добавляем дескрипторы $(S \to \cdot SSS, s_1, 1), (S \to \cdot SS, s_1, 1), (S \to \cdot a, s_1, 1)$.



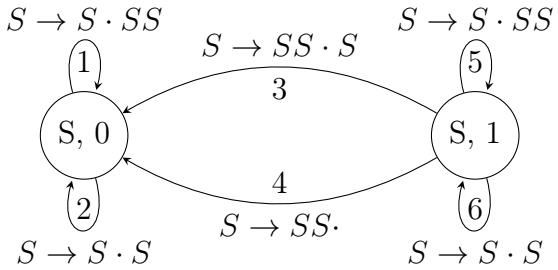
6. При обработке $(S \to S \cdot S, s_0, 1)$ образовываем ребро 4, новых дескрипторов не добавляется.



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

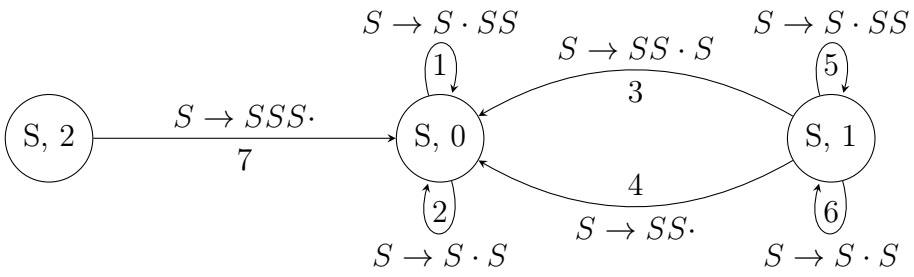


8. Обработка дескриптора $(S \to \cdot SS, s_1, 1)$ аналогична шагу 3 с добавлением петли 6.

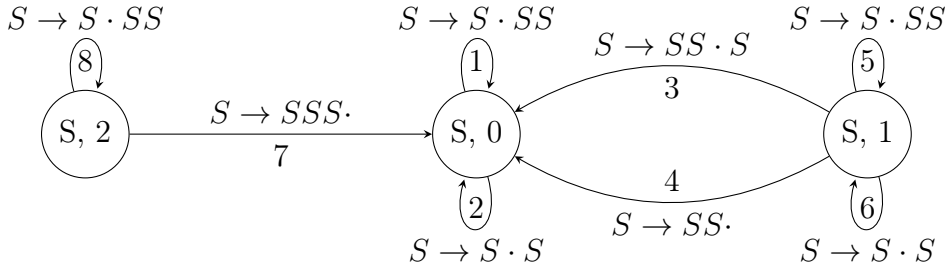


9. При обработке $(S \to \cdot a, s_1, 1)$ мы распознаем терминал a на позиции 1 и, возвращаясь по ребрам 3 и 4, добавляем дескрипторы $(S \to SS \cdot S, s_0, 2), (S \to SS\cdot, s_0, 2)$, а также, возвращаясь по петлям 5 и 6, добавляем дескрипторы $(S \to S \cdot SS, s_1, 2), (S \to S \cdot S, s_1,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

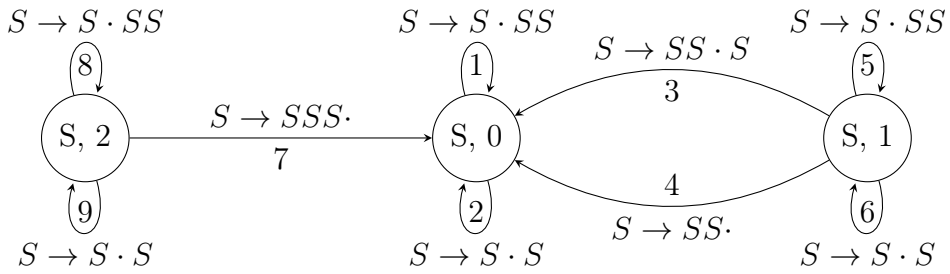
10. При обработке $(S \to SS \cdot S, s_0, 2)$ образовываем узел $s_2 = (S, 2)$ с исходящим ребром 7 и добавляем дескрипторы $(S \to \cdot SSS, s_2, 2), (S \to \cdot SS, s_2, 2), (S \to \cdot a, s_2, 2)$.



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 [1].

13.4 Алгоритм вычисления КС запросов на основе GLL

GLL довольно естественно обобщается на граф [33]: 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 [33]. Существует ещё одно обобщение нисходящего синтаксического анализа для решения задачи КС достижимости [47], 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 обрабатывается другими алгоритмами.

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

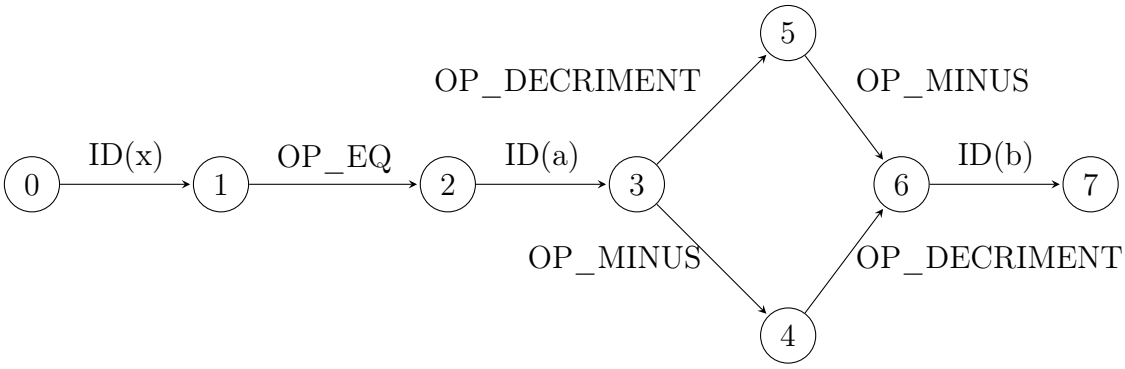
4. Использует существенно более тяжеловесные стуктуры данных и плохо распараллеливается (на практике). Как следствие, при решении задачи достижимости для всех пар путей проигрывает алгоритмам на основе линейной алгебры.

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 выглядеть 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. Данная идея рассматривается в работе [61].

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.

Глава 14

Алгоритм на основе восходящего анализа

В данном разделе будут рассмотрены алгоритмы восходящего синтаксического анализа LR-семейства, в том числе Generalized LR (GLR). Также будет рассмотрено обобщение алгоритма GLR для решения задачи поиска путей с контекстно-свободными ограничениями в графах.

14.1 Восходящий синтаксический анализ

Существует большое семейство LR(k) алгоритмов — алгоритм восходящего синтаксического анализа. Основная идея, лежащая в основе семейства, заключается в следующем: входная последовательность символов считывается слева направо с попутным добавлением в стек и выполнением сворачивания на стеке — замены последовательности терминалов и нетерминалов, лежащих наверху стека, на нетерминал, если существует соответствующее правило в исходной грамматике.

Как и в случае с LL, используется магазинный автомат, управляемый таблицами, построенными по грамматике. При этом, у LR анализатора есть два типа команд:

1. shift — прочитать следующий символ входной последовательности, положив его в стек, и перейти в следующее состояние;

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

А управляющая таблица выглядит следующим образом.

States	t_0	...	t_a	...	\$	N_0	...	N_b	...
...	...	...	...	...	...	...	...	...	...
10	...	...	s_i	...	r_k	...	...	j	...
...	...	...	...	...	acc	...	...	...	...

Здесь

- s_i — shift: перенести соответствующи символ в стек и перейти в состояние i.
- r_k — reduce(k): в стеке накопилась правая часть продукции k, пора производить свёртку.
- j — goto: выполняется после reduce. Сама по себе команда reduce не переводит автомат в новое состояние. Команда goto переведёт автомат в состояние j.
- acc — accept: разбор завершился успешно.

Если ячейка пустая и в процессе работы мы пропали в неё — значит произошла ошибка. Для детерминированной работы анализатора требуется, чтобы в каждой ячейке было не более одной команды. Есди это не так, то говорят о возникновении конфликтов.

- 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: $N \to w, M \to w$.

Принцип работы LR анализаторов следующий. 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 в управляющей таблице нет инструкции для текущего состояния автомата и текущего символа на входе, то завершаем разбор с ошибкой.
- Иначе выполняем одну из инструкций:
  - в случае acc — успешно завершаем разбор.
  - в случае shift — кладем на стек текущий символ входа, сдвигая при этом текущую позицию, и номер нового состояния. Переходим в новое состояние.
  - в случае reduce(k) — снимаем со стека 2l элементов: l состояний и l терминалов/нетерминалов (где l — длина правой части k-ого правила), кладём на стек нетерминал левой части правила. Тперь на вершине стека у нас нетерминал N_a, а следующий элемент — состояние i. Если в ячейке (i, N_a) управляющей таблицы лежит состояние j, то кладём его на вершину стека. Иначе завершаемся с ошибкой.

Разные алгоритмы из LR-семейства строят таблицы разными способами и, соответственно, могут избегать тех или иных конфликтов. Рассмотрим некоторых представителей.

14.1.1 LR(0) алгоритм

Данный алгоритм самый "слабый" из семейства: разбирает наименьший класс языков. Для построения используются LR(0) пункты.

Определение 14.1.1. LR(0) пункт (LR(0) 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): $A \to \alpha \cdot \beta$, где $A \to \alpha\beta$ — правило грамматики. □

Состояние LR(0) автомата — множество LR(0) пунктов. Для их построения используется операция *closure* или *замыкание*.

Определение 14.1.2. $closure(X) = closure(X \cup \{M \to \cdot\gamma \mid N_i \to \alpha \cdot M\beta \in X\})$ □

Определение 14.1.3. Ядро — исходное множество пунктов, до применения к нему замыкания. □

Для перемещения точки в пункте используется функция *goto*.

Определение 14.1.4. $goto(X, p) = \{N_j \to \alpha p \cdot \beta \mid N_j \to \alpha \cdot p\beta \in X\}$ □

Теперь мы можем построить LR(0) автомат. Первым шагом необходимо расширить грамматику: добавить к исходной грамматике правило вида $S' \to S\$$, где S — стартовый нетерминал исходной грамматики, S' — новый стартовый нетерминал (не использовался ранее в грамматике), \$ — маркер конца строки (не входил в терминальный алфавит исходной грамматики).

Далее строим автомат по следующим принципам.

- Состояния — множества пунктов.
- Переходы между состяниями осуществляются по символам грамматики.
- Начальное состояние — $closure(\{S' \to \cdot S\$\})$.
- Следующее состояние по текущему состоянию X и смволу p вычисляются как $closure(goto(X, p))$

Управляющая таблица по автомату строится следующим образом.

- *acc* в ячейку, соответствующую финальному состоянию и \$
- s_i в ячейку (j, t), если в автомате есть переход из состояния j по терминалу t в состояние i
- i в ячейку (j, N), если в автомате есть переход из состояния j по нетерминалу N в состояние i
- r_k в ячейку (j, t), если в состоянии j есть пункт $A \to \alpha\cdot$, где $A \to \alpha$ — k-ое правило грамматики, t — терминал грамматики

14.1.2 SLR(1) алгоритм

SLR(1) анализатор отличается от LR(0) анализатора построением таблицы по автомату (автомат в точности, как у LR(0). А именно, r_k добавляется в ячейку (j, t), если в состоянии j есть пункт $A \to \alpha\cdot$, где $A \to \alpha$ — k-ое правило грамматики, $t \in FOLLOW(A)$

14.1.3 CLR(1) алгоритм

Canonical LR(1), он же LR(1). Данный алгоритм является дальнейшим расширением SLR(1): к пунктам добавляются множества предпросмотра (lookahead).

Определение 14.1.5. Множество предпросмотра для правила P — терминалы, которые должны встретиться в выведенной строке сразу после строки, выводимой из данного правила. □

Определение 14.1.6. CLR пункт: $[A \to \alpha \cdot \beta, \{t_0, \ldots, t_n\}]$, где $t_0, \ldots, t_n$ — множество предпросмотра для правила $A \to \alpha\beta$. □

Определение 14.1.7. Пусть дана грамматика $G = \langle \Sigma, N, R, S \rangle$.

$$closure(X) = closure(X \cup \{[B \to \cdot\delta, \{FIRST(\beta t_0), \ldots, FIRST(\beta t_n)\}] \mid B \to \beta \in R, [A \to \alpha \cdot B\beta, \{t_0, \ldots, t_n\}] \in closure(X)\})$$

□

Функция *goto* определяется аналогично LR(0), автомат строится по тем же принципам.

При построении управляющей таблицы усиливается правило добавлеия команды *redice*. А именно, добавляем r_k в ячейку (j, t_i), если в состоянии j есть пункт $[A \to \alpha\cdot, \{t_0, \ldots, t_n\}]$, где $A \to \alpha$ — k-ое правило грамматики.

14.1.4 Примеры

Рассмотрим построение автоматов и таблиц для различных модификаций LR алгоритма.

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:

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

Пример 14.1.1. Пример ядра и замыкания.

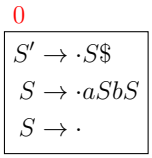
Возьмем правило 2 нашей грамматики, предположим, что мы только начинаем разбирать данное правило.

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

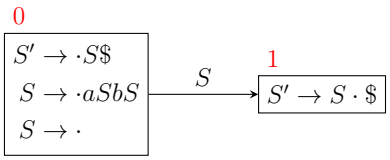
При замыкании добавятся ещё два item'а с правилами по выводу нетерминала 'S', поэтому получаем три item'а: $S' \to \cdot S\$$, $S \to \cdot aSbS$ и $S \to \cdot\varepsilon$

Пример 14.1.2. Пример построения LR(0)-автомата для нашей грамматики с применением замыкания.

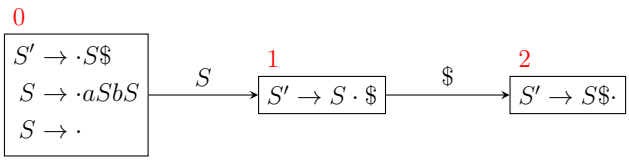
1. Добавляем стартовое состояние: item правила 0 и его замыкание (вместо item'а $S \to .\varepsilon$ будем писать $S \to .$).



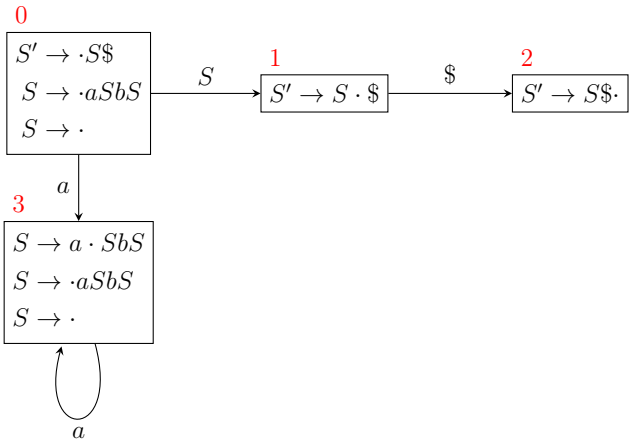
2. По 'S' добавляем переход из стартового состояния в новое состояние 1.



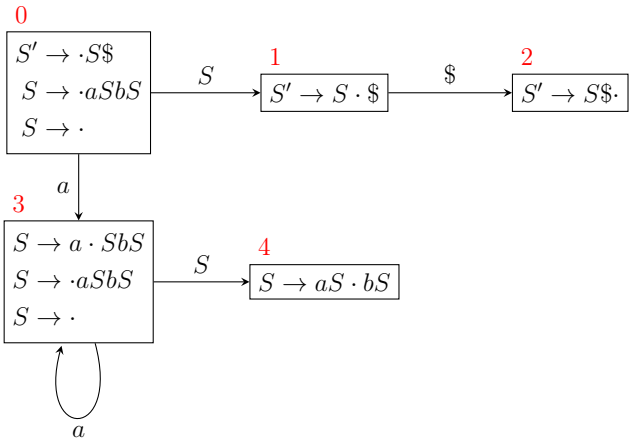
3. По '$' добавляем переход из состояния 1 в новое состояние 2.



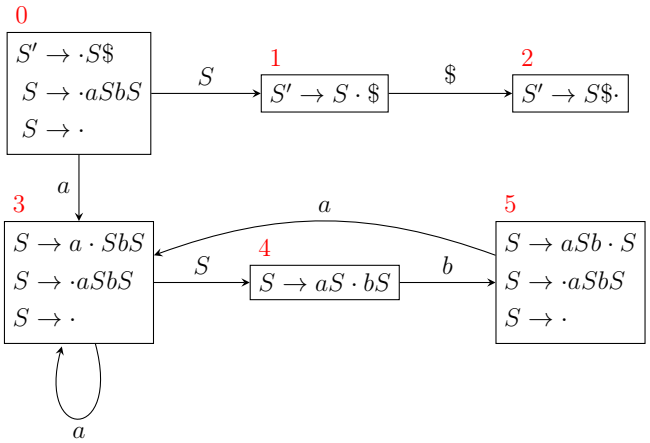
4. По 'a' добавляем переход из стартового состояния в новое состояние 3 и делаем его замыкание. Также добавляем переход по 'a' из этого состояния в себя же.



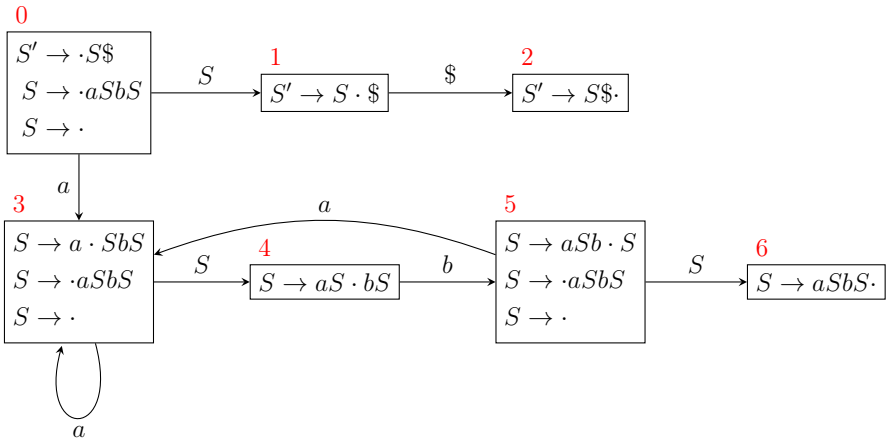
5. По ‘S’ добавляем переход из состояния 3 в новое состояние 4.



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.



Далее будем использовать этот автомат для построения управляющей таблицы.

Пример 14.1.3. Пример управляющей LR(0) таблицы.

	a	b	$	S
0	s_3, r_1	r_1	r_1	1
1			acc	
2	r_2	r_2	r_2	
3	s_3, r_1	r_1	r_1	4
4		s_5		
5	s_3, r_1	r_1	r_1	6
6	r_0	r_0	r_0	

Как видим, в данном случае в таблице присутствуют shift-reduce конфликты. В случае, когда не удаётся построить таблицу без конфликтов, говорят, что грамматика не LR(0).

□

Пример 14.1.4. Пример управляющей LR(1) таблицы. Автомат тот же, однако команды *reduce* расставляются с использованием FOLLOW.

$$FOLLOW_1(S) = \{b, \$\}$$

	a	b	$	S
0	s_3	r_1	r_1	1
1			acc	
2				
3	s_3	r_1	r_1	4
4		s_5		
5	s_3	r_1	r_1	6
6		r_0	r_0	

В данном случае в таблице отсутствуют shift-reduce конфликты. То есть наша грамматика SLR(1), но не LR(0).

□

Пример 14.1.5. 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.

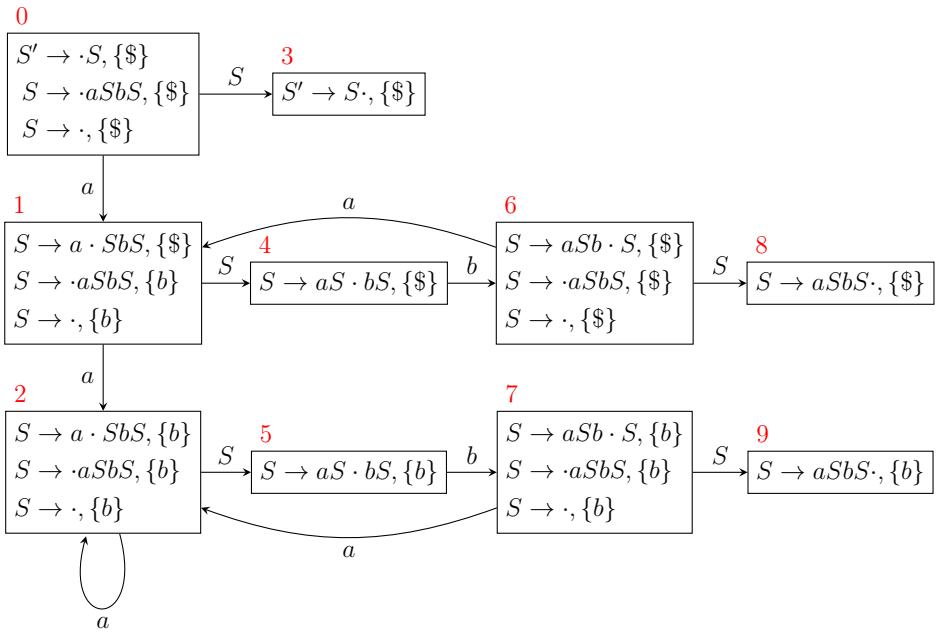
Вход:	a	b	a	b	$					
Стек:	0	a	3	S	4	b	5	S	6	

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		

□

Пример 14.1.6. Пример CLR автомата.



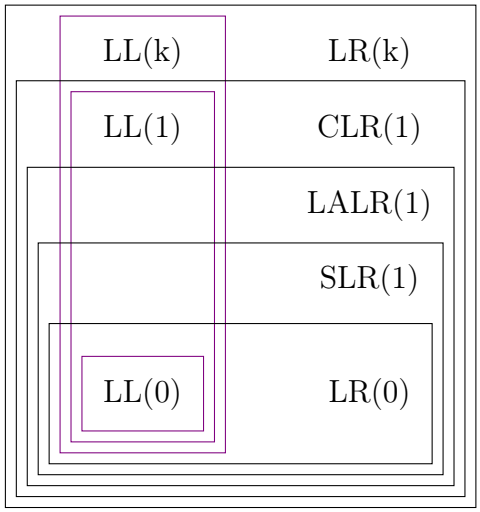
□

Существуют и другие модификации, например LALR(1)

На практике конфликты стараются решать ещё и на этапе генерации. Прикладные инструменты 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).

14.1.5 Сравнение классов LL и LR

Иерархию языков, распознаваемых различными классами алгоритмов, можно представить следующим образом.



Из диаграммы видно, что класс языков, распознаваемых LL(k) алгоритмом уже, чем класс языков, распознаваемый LR(k) алгоритмом, при любом конечном k. Приведём несколько примеров.

1. $L = \{a^m b^n c \mid m \geq n \geq 0\}$ является LR(0), но для него не существует LL(1) грамматики.
2. $L = \{a^n b^n + a^n c^n \mid n > 0\}$ является LR, но не LL.
3. Больше примеров можно найти в работе Джона Битти [13].

14.2 GLR и его применение для КС запросов

Алгоритм LR довольно эффективен, однако позволяет работать не со всеми КС-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 множественные 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) [70]. Далее мы рассмотрим общий принцип работы, проиллюстрируем его с помощью примера, а также рассмотрим модификации GLR.

14.2.1 Классический GLR алгоритм

Впервые GLR парсер был представлен Масару Томитой в 1987 [70]. В целом, алгоритм работы идентичен LR той разницей, что управляющая таблица модифицирована таким образом, чтобы допускать множественные значения в ячейках. Интерпретатор автомата изменён соответствующим образом.

Для того, чтобы избежать дублирования информации при обработке неоднозначностей, 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 дуг: таким образом реализуется то объединение одинаковых состояний и ветвление в случае неоднозначности.

Пример 14.2.1. Рассмотрим пример GLR разбора с использованием GSS.

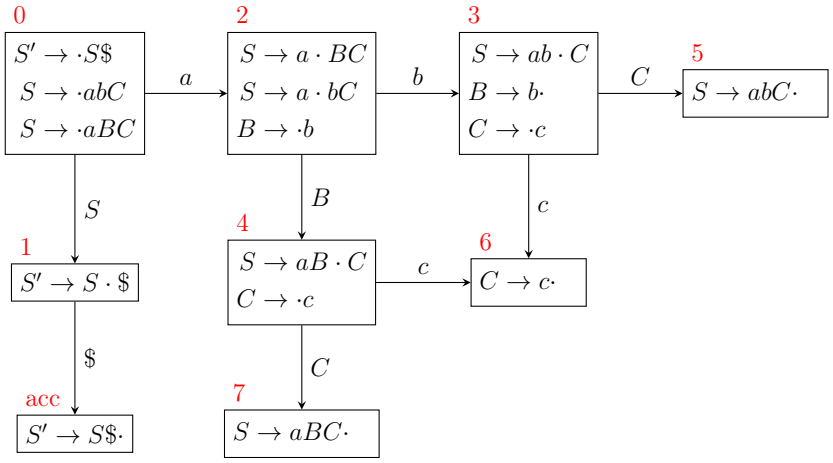
Возьмем грамматику G следующего вида:

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	s_2					1	
1				acc			
2		s_3			4		
3			s_6, r_3			5	
4			s_6			7	
5				r_1			
6				r_4			
7				r_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: v_0 (0)

2. Видим входной символ 'a', ищем соответствующую ему операцию в управляющей таблице — $shift$ 2, строим новый узел v_1:

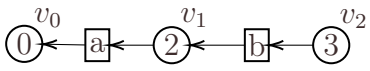
   Вход: | a | b | c | $ | GSS: v_0 (0) ← a ← v_1 (2)

3. Повторяем для символа ’*b*’, операции $shift$ 3 и узла v_2:

Вход:

a	b	c	$

GSS:

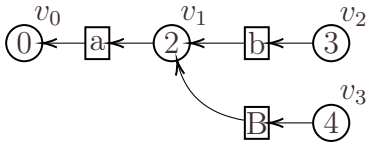


4. При обработке узла v_3 у нас возникает конфликт shift-reduce: s_6, r_3. Мы смотрим на вершины, смежные v_2, на управляющую таблицу и на правило вывода под номером 3 для поиска альтернативного построения стека. Находим *goto* 4 и строим вершину v_3 с соответствующим переходом по нетерминалу B из v_1 (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:

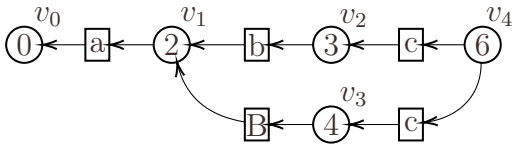


5. Читаем символ ’*c*’ и ищем в управляющей таблице переходы из состояний 3 и 4 (так как узлы v_2 и v_3 находятся на одном уровне, то есть были построены после чтения одного символа из входного слова). Таким переходом оказывается s_6 в обоих случаях, поэтому соединяем узел v_4 с обоими рассмотренными узлами:

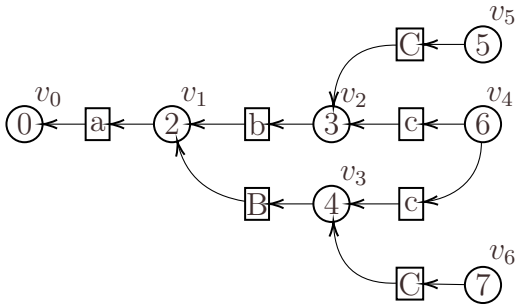
Вход:

a	b	c	$

GSS:

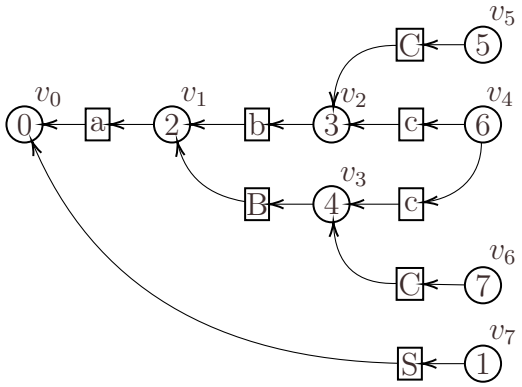


6. При обработке узла v_4 находим соответствующею 6-ому состоянию редукцию по правилу 4. Его правая часть содержит один символ ’*c*’, 2 вершины-символа с которым достижимы из v_4. Находим вершины-состояния, которые смежны с этими вершинами-символами и обрабатываем переходы по левой части правила 4. Такими переходами по нетерминалу C оказываются 5 и 7. Строим соответствующие им вершины v_5 и v_6:



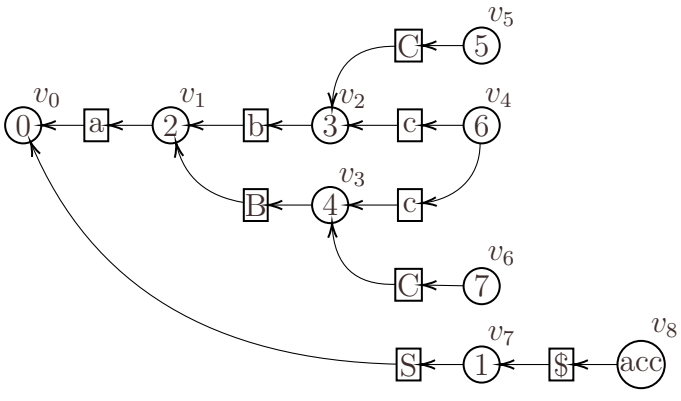
Вход: | a | b | c | $ | GSS:

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:

14.2.2 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 некоторое время после публикации Томита-парсера, Элизабет Скотт и Эндриан Джонстоун представили *RNGLR* (Right Nulled GLR) [59] — 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) [62]. 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 решения задачи поиска путей с КС ограничениями. Это происходит следующим образом: элементами во входной структуре теперь будем считать не позиции символа в слове, а вершины графа (то есть “позици” 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 [73].

Глава 15

Поиск путей с ограничениями в терминах многокомпонентных контекстно-свободных языков

В статтическом анализе кода — ещё одна аппроксимация. Например, On. Есть ли точно описываемые задачи?

Алгоритм на матрицах

Литература

[1] A. Afroozeh and A. Izmaylova. Faster, practical gll parsing. In B. Franke, editor, *Compiler Construction*, pages 89–108, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[2] M. R. Albrecht, G. V. Bard, and W. Hart. Efficient multiplication of dense matrices over GF(2). *CoRR*, abs/0811.1714, 2008.

[3] J. Alman and V. V. Williams. A refined laser method and faster matrix multiplication, 2020.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] N. Bansal and R. Williams. Regularity lemmas and combinatorial algorithms. In *2009 50th Annual IEEE Symposium on Foundations of Computer Science*, pages 745–754, 2009.

[9] J. Baras and G. Theodorakopoulos. Path problems in networks. In *Path Problems in Networks*, 2010.

[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] J. C. Beatty. Two iteration theorems for the ll(k) languages. *Theoretical Computer Science*, 12(2):193 – 228, 1980.

[14] 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.

[15] 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.

[16] 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.

[17] T. M. Chan. All-pairs shortest paths with real weights in $o(n^3/log\, n)$ time. *Algorithmica*, 50(2):236–243, Feb 2008.

[18] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5):2075–2089, 2010.

[19] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[20] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *SIAM Journal on Computing*, 11(3):472–492, 1982.

[21] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990.

[22] B. Courcelle and J. Engelfriet. *Graph Structure and Monadic Second-Order Logic*. Cambridge University Press, 2009.

[23] S. Dalton, N. Bell, L. Olson, and M. Garland. Cusp: Generic parallel algorithms for sparse matrix and graph computations, 2014. Version 0.5.0.

[24] D. Das, M. Koucký, and M. Saks. Lower bounds for combinatorial algorithms for boolean matrix multiplication, 2018.

[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] H. Ehrig, A. Habel, and H.-J. Kreowski. Introduction to graph grammars with applications to semantic networks. *Computers & Mathematics with Applications*, 23(6):557–572, 1992.

[29] N. El abbadi. An efficient storage format for large sparse matrices based on quadtree. *International Journal of Computer Applications*, 105:25–30, 11 2014.

[30] M. Elekes, A. Nagy, D. Sándor, J. B. Antal, T. A. Davis, and G. Szárnyas. A graphblas solution to the sigmod 2014 programming contest using multi-source bfs. In *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–7, 2020.

[31] R. W. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[32] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[33] 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.

[34] Y. Han. Improved algorithm for all pairs shortest paths. *Information Processing Letters*, 91(5):245–250, 2004.

[35] J. Hellings. Conjunctive context-free path queries. In *Proceedings of ICDT'14*, pages 119–130, 2014.

[36] J. Hellings. Path results for context-free grammar queries on graphs. *ArXiv*, abs/1502.02242, 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] J. Kepner and J. Gilbert. *Graph Algorithms in the Language of Linear Algebra.* Society for Industrial and Applied Mathematics, 2011.

[44] 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.

[45] 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.

[46] L. Lee. Fast context-free grammar parsing requires fast boolean matrix multiplication. *J. ACM*, 49(1):1–15, Jan. 2002.

[47] 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.

[48] 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.

[49] A. Okhotin. Conjunctive grammars. *Journal of Automata, Languages and Combinatorics*, 6(4):519–535, 2001.

[50] A. Okhotin. On the closure properties of linear conjunctive languages. *Theor. Comput. Sci.*, 299(1-3):663–685, 2003.

[51] 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.

[52] P. Pratikakis, J. S. Foster, and M. Hicks. Existential label flow inference via cfl reachability. In *SAS*, volume 6, pages 88–106. Springer, 2006.

[53] J. Rehof and M. Fähndrich. Type-base flow analysis: From polymorphic subtyping to cfl-reachability. *SIGPLAN Not.*, 36(3):54–66, Jan. 2001.

[54] J. G. Rekers. *Parser generation for interactive environments.* PhD thesis, Universiteit van Amsterdam, 1992.

[55] 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.

[56] 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.

[57] B. Roy. Transitivité et connexité. *Comptes Rendus Hebdomadaires Des Seances De L Academie Des Sciences*, 249(2):216–218, 1959.

[58] A. Schönhage. Partial and total matrix multiplication. *SIAM Journal on Computing*, 10(3):434–455, 1981.

[59] E. Scott and A. Johnstone. Right nulled glr parsers. *ACM Trans. Program. Lang. Syst.*, 28(4):577–618, July 2006.

[60] E. Scott and A. Johnstone. Gll parsing. *Electron. Notes Theor. Comput. Sci.*, 253(7):177–189, Sept. 2010.

[61] 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.

[62] E. Scott, A. Johnstone, and R. Economopoulos. Brnglr: A cubic tomita-style glr parsing algorithm. *Acta Inf.*, 44(6):427–461, Sept. 2007.

[63] H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191 – 229, 1991.

[64] P. Sevon and L. Eronen. Subgraph queries by context-free grammars. *Journal of Integrative Bioinformatics*, 5, 06 2008.

[65] 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.

[66] V. Strassen. Gaussian elimination is not optimal. *Numerische mathematik*, 13(4):354–356, 1969.

[67] T. Takaoka. A new upper bound on the complexity of the all pairs shortest path problem. *Information Processing Letters*, 43(4):195–199, 1992.

[68] 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.

[69] 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.

[70] M. Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13:31–46, 1987.

[71] 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.

[72] L. G. Valiant. General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.*, 10(2):308–315, Apr. 1975.

[73] 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.

[74] 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.

[75] S. Warshall. A theorem on boolean matrices. 1962.

[76] V. V. Williams and R. Williams. Subcubic equivalences between path, matrix and triangle problems. In *Proceedings of the 2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, FOCS '10, pages 645–654, Washington, DC, USA, 2010. IEEE Computer Society.

[77] N. Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of the ACM*, 20(11):822–823, 1977.

[78] 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.

[79] 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.

[80] H. Yu. An improved combinatorial algorithm for boolean matrix multiplication. In M. M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Automata, Languages, and Programming*, pages 1094–1105, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[81] 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.

[82] 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.

[83] 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.

[84] R. Zier-Vogel and M. Domaratzki. Rna pseudoknot prediction through stochastic conjunctive grammars. *Computability in Europe 2013. Informal Proceedings*, pages 80–89, 2013.

[85] 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.

[86] В. Арлазаров, Е. Диниц, М. Кронрод, and И. Фараджев. Об экономном построении транзитивного замыкания ориентированного графа. *Докл. АН СССР*, 194(3):487–488, 1970.

[87] Н. Вавилов. КОНКРЕТНАЯ ТЕОРИЯ ГРУПП, 2005. Дата доступа: 29 июня 2021 г.

[88] Н. Вавилов. КОНКРЕТНАЯ ТЕОРИЯ КОЛЕЦ, 2006. Дата доступа: 29 июня 2021 г.