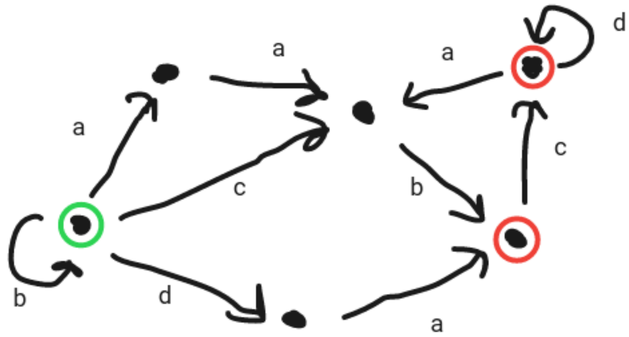
(a) ДКА с одним состоянием.



(b) Простой ДКА.

Рис. 1: Примеры ДКА. Зелёным выделены начальные состояния, красным – терминальные.

Интуиция

Здесь будет изложена базовая теория конечных автоматов, регулярных выражений и их связи.

Конечные автоматы, нестрого, – это сущность, которая имеет несколько состояний, в моменте находится в одном, и умеет переходить между состояниями. Самый простой пример – лифт. Он может находиться на n этажах, в моменте находится на одном этаже, и нажатием кнопки вы переводите его на другой этаж (забудем пока про время поездки).

Конечным автоматом не являются ни сущности, умеющие самостоятельно менять состояние (блокировка экрана по таймауту), ни сущности с бесконечным числом состояний (редактируемый текстовый документ).

Регулярные выражения, в жизни, – это способ проверить строку на соответствие паттерну. Или, проще, способ найти "вот что-то такое" в текстовом документе, который глазами просмотреть не получается.

Оказывается, что регулярные выражения математически эквивалентны конечным автоматам, а на практике работать с автоматами куда проще, чем с регулярными выражениями. Поэтому давайте наведём немного формализма.

Языки и DFA

Введём несколько определений, которые понадобятся в будущем.

Алфавит Σ – это некоторое конечное множество. Элементы этого множества мы будем называть символами или буквами. Назовём Σ^* все возможные последовательности символов этого алфавита (все "слова" алфавита).

Языком L будем называть любое подмножество Σ^* (любой набор слов). Регулярным языком, или языком, задаваемым конечным автоматом, будем называть язык... задаваемый конечным автоматом.

Детерминированным конечным автоматом (ДКА, deterministic finite automaton, DFA) назовём ориентированный граф со множеством вершин S, множеством рёбер E, удовлетворяющий следующим условиям:

- одна из вершин выделена и называется начальной;
- сколько-то вершин может быть выделено и названо конечными или терминальными (терминальных вершин может не быть вообще);
- над каждым ребром написан символ алфавита;
- для любых вершины и символа существует не больше одного ребра, выходящего из данной вершины и подписанного данным символом.

Пусть у нас есть слово $w = \alpha_1\alpha_2\ldots\alpha_n$, где α_i – символ алфавита. Назовём начальное состояние ДКА s_0, и будем обозначать состояния вообще s_i. Слово можно "скормить" конечному автомату, взяв начальное состояние и переходя по символам слова в том порядке, в котором они написаны: $s_0 \xrightarrow{\alpha_1} s_{i_1} \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_n} s_{i_n}$. Такая процедура закончится либо когда автомат сломается (перехода из данного состояния по данному символу нет), либо когда слово закончится. Если слово закончилось, и последнее состояние в цепочке переходов является терминальным, то будем говорить, что ДКА *распознаёт* данное слово.

Например, тривиальный ДКА с рисунка 1a распознаёт слова “” (пустая строка), “a”, “aa”... ДКА с рисунка 1b распознаёт слова “da”, “bcb”, “aab” и множество других. Но он не распознаёт, например, слова “dc” или “aa”: на первом слове автомат сломается на втором переходе, на втором слове автомат не сломается, но закончит не в терминальном состоянии.

Множество слов, которое распознаёт данный ДКА, будем называть языком, задаваемым этим ДКА.

Регулярные выражения, упомянутые в начале, проверяют строки (слова) на соответствие паттерну. То есть, на самом деле, они выделяют из всего множества слов подмножество удовлетворяющих паттерну – и этим определяют язык. Как говорят, регулярный язык. Более того, ДКА и регулярные выражения определяют одно и то же множество языков: для любого регулярного выражения существует ДКА, определяющий тот же язык, поэтому для проверки соответствия строки паттерну достаточно проверить, распознаётся ли эта строка соответствующим ДКА. Это основа работы с регулярными выражениями.

Построение ДКА, соответствующего заданному регулярному выражению – это отдельная задача, которую мы будем рассматривать ниже. Но до этого нам понадобится ещё несколько определений и отступлений на тему реализации описанной теории.

DFA: implementation details

Детерминированный конечный автомат – это граф, и все способы хранения графов применимы и здесь. Однако удобнее всего хранить ДКА в виде матрицы переходов и вектора терминальных состояний.

В матрице переходов T число состояний строк и размер алфавита столбцов. Для каждого перехода $s_1 \xrightarrow{\alpha_1} s_2$ в матрице переходов есть запись $T[s_1][\alpha_1] = s_2$.

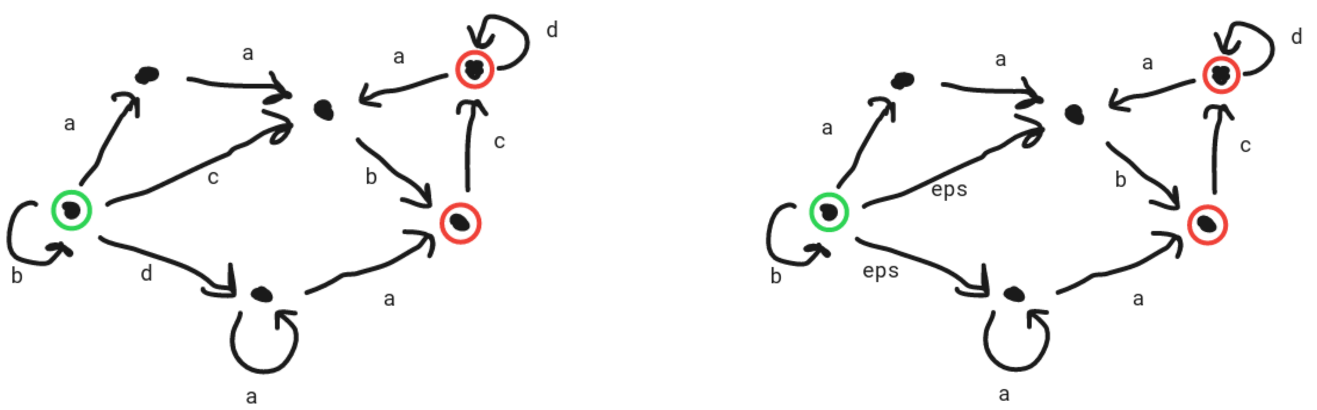
Стоит обратить отдельное внимание на то, что из каких-то состояний может не быть перехода для некоторых символов. Такие ситуации в матрице нужно обрабатывать отдельно, и в процессе распознавания слова не пытаться идти по несуществующим переходам.

Вектор терминальных состояний удобно сделать длиной в число состояний, записать 0 по индексам состояний, не являющихся терминальными, и 1 для терминальных состояний.

Таких структур должно хватить, чтобы быстро проверять, распознаёт ли ДКА данное слово (быстро = за время порядка длины слова).

NFA и ϵ-NFA

Чтобы привести регулярные выражения к DFA, нам понадобится несколько дополнительных понятий.



(a) НКА: из нижнего узла выходит два символа a.

(b) ϵ-НКА.

Рис. 2: Примеры НКА и ϵ-НКА. ϵ на рисунках обозначено как eps.

Недетерминированный конечный автомат (НКА, nondeterministic finite automaton, NFA) – это детерминированный конечный автомат без ограничения на число переходов из одного состояния по одному символу. То есть, всё ещё ориентированный граф с начальной и, возможно, терминальными вершинами и подписями над рёбрами, но теперь из одной вершины может выходить два и более ребёр с одинаковыми подписями.

Распознавание слова НКА происходит аналогично ДКА, только в случае нескольких переходов нужно выбрать один. Если существует хотя бы один путь от начального состояния к терминальному по символам слова, то слово распознаваемо. Например, НКА на рис. 2a распознаёт слова “da”, “daa”, “bbdaaaacdab” и так далее (см. нижнюю ветку).

ϵ-недетерминированный конечный автомат (ϵ-НКА, ϵ-NFA) – это НКА, в котором помимо символов алфавита есть специальный символ ϵ. Переход по нему разрешён в любой момент и, условно, не “стоит” символа слова. В ДКА и НКА при распознавании слова “abc” мы совершали ровно три перехода, если автомат не ломался раньше; в ϵ-НКА можно идти по пути “ϵa$\epsilon\epsilon$bcϵ”, и этот путь будет считаться путём “abc”.

Например, ϵ-НКА на рис. 2b распознаёт слова “a” и “b”. ДКА и НКА на рис. 1b и 2a такие слова не распознавали, потому что нельзя было добраться до конечного состояния менее чем за два перехода.

NFA: implementation details

Будем рассматривать конкретно задачу распознавания слова конечным автоматом. В ДКА по состоянию и символу было однозначно известно конечное состояние; в НКА так уже не работает, один символ может вести в несколько конечных состояний. Поэтому придётся заменить промежуточные состояния НКА на наборы состояний: все возможные состояния, в которые можно было прийти по данному символу из предыдущего набора состояний. Например, возможна такая цепочка переходов: $\{s_0\} \xrightarrow{\alpha_1} \{s_1, s_2\} \xrightarrow{\alpha_2} \{s_1, s_3, s_5, s_9\}$...

В матрице переходов возникает та же проблема: начальное состояние и символ теперь определяют набор конечных состояний.

Все эти наборы нужно как-то хранить. Очевидно, можно просто запихнуть всё в вектора, но есть более элегантное решение.

Давайте сначала предположим, что число состояний меньше 64. Возьмём 64-битное число и зашифруем в него набор состояний по следующей логике: если в наборе есть состояние i, то i-ый бит числа поставим 1, а если состояния нет, то соответствующий бит сделаем 0. Тогда в каждой ячейке матрицы переходов снова будет храниться число, и текущий набор состояний тоже будет числом. Более того, сложение наборов состояний теперь можно делать битовым or.

Если число состояний больше 64, то придётся хранить несколько чисел: в первом числе первые 64 состояния, во втором с 65 по 128 и так далее. Тем не менее, описанная выше кодировка позволит существенно уменьшить константу как в требуемой памяти, так и во времени исполнения алгоритма.

Кроме того, нужно будет поменять условие успешного распознавания: для НКА слово успешно прочитано, если автомат не сломался в середине процесса, и в конечном наборе состояний есть хотя бы одно терминальное состояние.

ϵ-NFA: implementation details

Для ϵ-НКА верны все замечания, которые верны для обычных НКА, но ситуация осложняется символом ϵ. Чтобы справиться с ним, введём понятие ϵ-замыкания.

ϵ-замыкание набора состояний S’ для ϵ-НКА – это набор состояний, включающий в себя сам S’ и все состояния, до которых можно добраться из S’, двигаясь только по ϵ-переходам.

Если ϵ-замыкание предварительно сделать над каждым состоянием, то задача распознавания ϵ-НКА сводится к задаче распознавания обычным НКА.

Альтернативно, можно в процессе распознавания слова после каждого перехода делать ϵ-замыкание (включая последний переход), тогда сам переход будет аналогичен переходу в НКА. Но такой способ может занимать больше времени.

Регулярные выражения

pass

Сведение регулярного выражения к ϵ-NFA

pass

Сведение ϵ-NFA к DFA

pass

Минимизация DFA

ДКА, полученный из ϵ-НКА наверняка будет большим, но не обязательно будет оптимальным с точки зрения числа состояний. Возникает задача поиска ДКА, эквивалентного данному, с минимальным возможным числом состояний. (Эквивалентные ДКА – те, которые задают один и тот же язык.)

Существует два класса состояний, которые можно удалить из ДКА: недостижимые и неразличимые. Недостижимые – те, которых нельзя достигнуть из начального состояния ни по каким переходам. Неразличимые – те, из которых переход по любому слову заканчивается одинаково (распознаванием слова или не распознаванием).

Удаление недостижимых состояний – достаточно простая задача, см. implementation details. С неразличимыми ситуация несколько сложнее: как можно понять, что из двух разных состояний абсолютно все возможные слова распознаются одинаково? Слов ведь бесконечное количество. Оказывается, есть у регулярных языков хорошая, конечная характеристика – число производных языка.

Назовём производной языка L по строке α множество строк $L_\alpha = \{\beta | \alpha\beta \in L\}$. То есть, множество всех строк таких, что если к ним в начало приставить α, то получится слово из L. Обратите внимание на слово "всех": производная всегда однозначна определена и содержит продолжения всех слов L, начинающихся на α.

Пример. На рис. 1a представлен ДКА, который определяет язык $L = \{\epsilon, a, aa, aaa, \dots\}$ (первый элемент – пустая строка). Производная этого языка по символу a будет $L_a = \{\epsilon, a, aa, aaa, \dots\}$. Пустая строка исходного языка не начиналась на символ a, и поэтому не попала в производную; символ a исходного языка перешёл в пустую строку производной, и так далее. Нетрудно заметить, что $L_a = L$, и более того, производная L по строке из любого количества символов a тоже будет равна L.

Стоит также рассмотреть физический смысл этой производной. Пусть у нас есть ДКА, задающий язык L, и мы пытаемся распознать им строку $\alpha\beta = \alpha_1 \dots \alpha_n \beta_1 \dots \beta_k$. Пусть под действием α ДКА переходит из начального состояние s_0 в какое-то другое s_{i_n}. Тогда язык L_α содержит все слова, которые можно распознать этим ДКА, начиная из состояния s_{i_n}. В самом деле: в языке L содержались все слова, которые распознавал автомат; мы от всех вида $\alpha\beta$ отрезали α и под его действием перешли в состояние s_{i_n}. Значит, строка β должна распознаваться из состояния s_{i_n}, иначе исходная строка $\alpha\beta$ не распознавалась бы этим автоматом.

То есть, производная языка соответствует смещению начального состояния в какое-то другое (достижимое) состояние и характеризует все слова, которые можно из этого состояния распознать. На этом фоне могут стать интуитивно понятны следующие теоремы.

- Язык является регулярным если и только если количество его различных производных конечно.
- Количество непустых различных производных языка равно количеству состояний в минимальном по количеству состояний ДКА, описывающем данный язык.

Их доказательство сводится к корректному повторению одних и тех же слов в разном порядке. Пусть есть ДКА, оригинально названный A, и задаваемый им непустой язык L, тогда:

- Каждое достижимое состояние задаёт одну и только одну производную языка L. В самом деле, поскольку состояние достижимо, существует строка α, по которой можно до него добраться; тогда это состояние задаёт производную L_α. Все строки в этой производной – строки, которые A распознает из данного состояния; не может существовать двух различных множеств строк, которые автомат распознаёт из данного состояния, это бы означало, что существует строка, которую автомат одновременно распознаёт и не распознаёт. Что бред. Поэтому задаваемая производная единственна.

- Каждая производная языка L, кроме пустой, задаёт хотя бы одно состояние A. В самом деле, пусть в производной L_α содержится строка β. Тогда строка $\alpha\beta$ распознаётся автоматом, строка α переводит автомат из начального состояния в какое-то, и именно его я предъявлю для доказательства утверждения.

- Если язык регулярен, то число его различных производных конечно. В самом деле, регулярный язык задаётся автоматом с конечным количеством состояний, и каждое состояние задаёт только одну производную. Производные могут повторяться, так что их станет меньше, чем состояний, но их не может стать больше, чем состояний.

- Если число производных языка конечно, то он может быть задан автоматом. Этот автомат можно сконструировать руками. Возьмём все непустые производные языка и перенумеруем их. Это будут состояния нашего автомата. Начальное состояние соответствует производной языка по пустой строке (равной исходному языку). Конечные состояния соответствуют производным, в которых есть пустая строка – это значит, что из такого состояния можно никуда не ходить и успешно закончить распознавание. Переходы восстанавливаются следующим образом: для каждой производной L_α и каждого символа алфавита a посчитаем производную $L_{\alpha a}$. Это будет соответствовать переходу из состояния, соответствующего L_α, по символу a. Если получилась пустая производная, переход по символу a не добавляем; иначе должна была получиться другая производная языка L, которой соответствует какое-то состояние. Рисуем переход в это состояние. На этом ДКА восстановлен.

- Из второй теоремы: по конечному количеству производных я могу предоставить ДКА с числом состояний, равным числу производных. Вот прям в прошлом пункте построил.

- Не существует автомата, задающего язык L, число состояний которого меньше числа непустых различных производных L. Докажите в качестве упражнения аналогично предыдущим пунктам.

Отличные новости! Мы нашли те объекты, которых конечное количество, и которые напрямую связаны с числом состояний в минимальном по числу состояний ДКА.

Отвратительные новости: сами эти объекты по большей части бесконечны, и попытки записать в память компа что-то бесконечное на моей памяти хорошо не заканчивались.

Заметим ещё связь с началом рассуждения: "неразличимые состояния" – то же, что и "состояния, задающие одинаковые производные". Ещё такие состояние можно назвать эквивалентными. Эквивалентность, как мы выяснили, включает в себя равенство бесконечных объектов; давайте обрежем её до конечных значений.

Будем называть состояния k-эквивалентными, если из них переход по любому слову длины не больше k заканчивается одинаково (распознаванием или не распознаванием). Или, что то же, одинаковы сужения производных языка, соответствующих этим состояниям, на слова длины не больше k.

0-эквивалентность достаточно проста. Ходить вообще никуда нельзя, мы остаёмся в состоянии, с которого начали, и ждём вердикта. Если мы начали в терминальном состоянии, то распознавание пройдёт успешно; если начали не в терминальном состоянии, то распознавания не будет. Так что все терминальные состояния 0-эквивалентны друг другу, и все не терминальные состояния 0-эквивалентны друг другу.

Как понять, что состояния k-эквивалентны друг другу? Предположим, что мы знаём всё про (k-1)-эквивалентность состояний, и попытаемся что-то понять про k-эквивалентность пары состояний, s_1 и s_2. Во-первых, чтобы быть k-эквивалентными, состояния должны быть (k-1)-эквивалентными, иначе уже есть примеры строк длины менее k, на которых распознавание из этих состояний даёт разные результаты.

Во-вторых, неожиданно, $k = 1 + (k - 1)$. Можно сместиться из исходных состояний на один, одинаковый символ, и посмотреть на (k-1)-эквивалентность полученных состояний. Если они окажутся (k-1)-эквивалентными, то все строки длины не больше k, начинающиеся на данный символ, приведут к одинаковому результату распознавания. Если это окажется верно для всех символов алфавита, и сами состояния будут (k-1)-эквивалентными, то состояния будут k-эквивалентными.

Поскольку мы знаем всё про 0-эквивалентность и про переход от (k-1)-эквивалентности к k-эквивалентност мы можем вычислить эквивалентность любого порядка. Что будет не очень полезно, если мы не сможем остановиться в какой-то момент и сказать, что эквивалентность текущего порядка – то же, что и просто эквивалентность.

Если внимательно присмотреться к алгоритму вычисления k-эквивалентности, то можно заметить, что он зависит только от (k-1)-эквивалентности. Ни от самого k, ни от (k-2)-эквивалентности, ни от чего. А это означает, что если после вычисления k-эквивалентность оказалась такой же, как и (k-1)-эквивалентность, то и (k+1), и все дальнейшие эквивалентности будут такими же: каждая зависит только от предыдущей, и алгоритм на предыдущей эквивалентности даёт точно такую же эквивалентность. Поэтому в момент, когда (k-1)- и k-эквивалентность совпали, можно считать, что мы разбили на классы эквивалентности все состояния.

Число этих классов эквивалентности будет равно как числу непустых производных языков, так и числу состояний в минимальном ДКА, описывающем данный язык. Признав каждый класс состоянием, можно построить минимальный ДКА, что и было целью этого параграфа.

Минимизация DFA: implementation details

Разберёмся сначала с удалением недостижимых состояний. Их нужно найти и удалить. Поиск достаточно прост: любой метод обхода вершин графа подойдёт для поиска всех достижимых вершин (см., например, обход в ширину или обход в глубину). Убрав из всех вершин все достижимые вершины, получим все недостижимые. Это можно сделать множеством способов; я опишу самый нетребовательный из линейных по числу состояний. Создадим массив с длиной равной числу состояний и заполним его нулями; в элементах, соответствующих достижимой вершине, поставим 1. Теперь в этом массиве нулями отмечены недостижимые вершины. За один проход по массиву их можно вытащить в отдельный вектор, если хочется.

Удаление следует производить аккуратно: нужно не только удалить состояние из матрицы переходов и вектора терминальных состояний, но и поправить матрицу переходов. Если под удалением понимать буквально удаление строки из матрицы и смещение всех следующих состояний на один к началу вектора, то у некоторых состояний автомата поменяются индексы; их нужно будет поправить. В худшем случае это займёт (число состояний удалений) * (число состояний сдвигов + число состояний * размер алфавита поправок). Обозначив число состояний автомата n, а размер алфавита l, получим сложность $O(n^2l)$. Алгоритм про неразличимые состояния имеет такую же сложность, так что заниматься оптимизацией на этом этапе смысла нет, и я не буду углубляться. В качестве самостоятельного упражнения попробуйте придумать способ удаления недостижимых состояний за $O(nl)$.

Теперь займёмся неразличимыми состояниями. А точнее, пересечением эквивалентностей.

Что такое эквивалентность программно? Во-первых, слово "эквивалентность" упомянуто не просто так: отношение неразличимости является рефлексивным, симметричным и транзитивным, то есть эквивалентностью. Нас больше волнует транзитивность: если $s_1 \overset{k}{\sim} s_2$ и $s_2 \overset{k}{\sim} s_3$, то $s_1 \overset{k}{\sim} s_3$. Так образуются "классы эквивалентности" состояний, в которых каждое состояние k-эквивалентно всем остальным.

Представить классы эквивалентности можно вектором, в котором i-ый элемент – класс эквивалентности i-ого состояния (например, состояние 0, 2 и 3 в классе эквивалентности 0, состояние 1 в классе эквивалентности 1, и так далее).

Дальше в алгоритме говорится примерно следующее: если два состояния принадлежат одному классу (k-1)-эквивалентности, и для всех символов переход по символу приводит в один класс (k-1)-эквивалентности, то эти два состояния принадлежат одному классу k-эквивалентности.

Для начала, нужно для каждого состояния сделать вектор следующего вида: (класс эквивалентности состояния; класс эквивалентности состояния, получившегося при переходе по первому символу; ... второму символу; ...; ...последнему символу). Если два таких вектора равны, то состояния принадлежат одному новому классу эквивалентности.

Конечно, сравнивать все вектора попарно будет слишком долго; нужно просто запихнуть их в один массив и запустить на нём поразрядную сортировку. Она сгруппирует одинаковые вектора, и код будет иметь вид “если этот вектор равен предыдущему, то написать ему тот же класс эквивалентности; иначе создать новый класс эквивалентности и поместить в него только текущий вектор”. Это линейное количество сравнений вместо квадратичного.

Определение классов эквивалентности следующего порядка, включая сортировку, будет занимать $O(nl)$ времени; провести эту процедуру придётся максимум n раз, потому что за одну итерацию добавляется хотя бы один класс, и классов не может быть больше, чем состояний. Так что общее время работы алгоритма $O(n^2l)$.