Суффиксный автомат

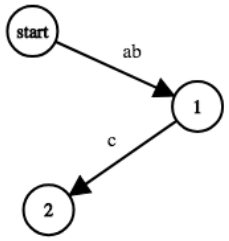
Филипп Грибов

14 ноября 2018 г.

1 Автомат

Представим себе ориентированный граф, в котором на каждом ребре записано по букве. Так же выделим одну из вершин и назовём её стартовой. Замечаем, что в таком графе каждый путь от стратовой вершины до какой-то будет задан определённой строкой. Рассмотрим следующий пример:

В автомате 3 вершины start, 1, 2, есть рёбро от start к 1, на котором написаны буквы «a» и «b» и есть ребро от 1 к 2, на котором написано «c». Заметим, что в таком автомате от start есть 5 путей: пустой путь, путь через «a» в 1, путь через «b» в 1, путь через «a» а потом через «c» в 2, пусть через «a» а потом через «c» в 3. Т.е возможные пути заданы строками «», «a», «b», «ac», «bc».



Таким образом в автомат можно передать строки и итереруясь по символам строки можно переходить из одной вершины в другую. В предыдущем примере в автомат можно передать строку «ac» и после этого мы окажемся в вершине 2. Но строку «bd» в предыдущий автомат передать нельзя, так как из 1 нет ребер по символу «d».

1.1 Основные понятия

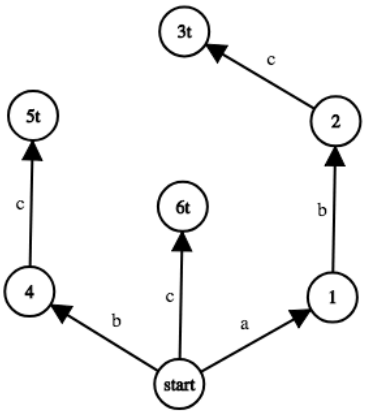
В автомате вершины принято называть **состояниями**, а рёбра — **переходами**.
Автомат называется **детерминированным**, если из любого состояния не существует двух переходов по одинаковым символам. Это означает, что для каждой строки есть не более одного пути, соответствующего ей и передав строку автомату мы однозначно знаем, куда по ней мы перейдём, если путь по ней существует.
Автомат называется **конечным**, если в автомате нет циклов, т.е. в нём не существует бесконечных по длине путей (возможно самопересекающихся).
Некоторые вершины автомата будут помечены как **терминальные**. Считается, что **автомат принял строку** если существует путь, соответствующий этой строке и в конце, пройдя по пути, мы окажемся в терминальной вершине.

2 Суффиксный автомат

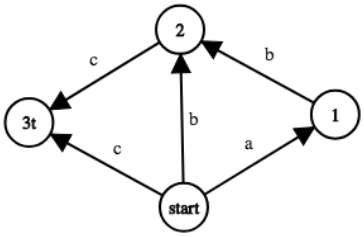
Суффиксный автомат это конечный детерминированный автомат, принимающий все суффиксы данной строки и только их. Если это определение у вас вызвало ужас и абсолютное непонимание происходящего, посмотрите все термины в предыдущем разделе. Далее все эти страшные слова мы почти не будем использовать.

На следующей странице показаны примеры суффиксных автоматов

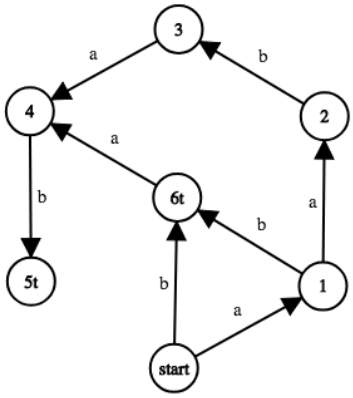
Бор, построенный на всех суффиксах строки нам вполне подойдёт. Для примера возьмём строку «abc»



Для той же строки можно построить суффиксный автомат гораздо меньшего размера:



Для строки «aabab» можно обойтись всего 7 вершинами. Запомните эту строку, её удобно использовать для дебага.



А теперь перейдём к алгоритму, строещему суффиксный автомат размера $O(n)$, где n — длина строки. Теория будет сильно отличаться от общепринятой в сторону большей понятности. Но для начала введём некоторые дополнительные ограничения и заметим некоторые вещи.

Для каждой вершины в автомате есть какой-то набор строк, пути по которым заканчиваются в этой верщине. Тогда будем поддерживать следующее правило. **Для каждой вершины все строки, ведущие в неё должны являться суффиксами друг друга.** Причём эти суффиксы должны являться подряд идущими по длине. Т.е если s — максимальная по длине строка, которая ведёт в вершину и если k — длина s, то в вершину ведут строки $s[0:k], s[1:k], s[2:k], s[3:k], \ldots, s[l:k]$. Итого в вершину будет вести $l+1$ строка, причём все строки будут являтся суффиксами s длины хотя-бы $k-l$. Соответственно наборы («z») или («abc», «bc», «c») или («abacaba», «bacaba», «acaba») могут являться полным набором строк, ведущих в вершину, а наборы («abacaba», «abacab») или («abcd», «cd», «d») или («abacab», «bacab») не могут являться полным набором строк, ведущих в вершину.

Будем говорить, что **вершине соответствует максимальная строка, которая в неё ведёт**. Т.е. если в вершину ведёт единственная строка («z»), то этой вершине соответствует «z», если в вершину ведут строки («abc», «bc», «c»), то вершине соответствует «abc», если в вершину ведут строки («abacaba», «bacaba», «acaba»), то вершине соответствует строка «abacaba».

По аналогии с Ахо-Карасиком, мы можем ввести понятие **суффиксная ссылка**. Как мы уже знаем, вершина A принимает строки $s[0:k], s[1:k], s[2:k], s[3:k], \ldots, s[l:k]$ (Суффиксы s длиной $k, k-1, \ldots, k-l$). А где-то должна быть вершина B, в которую ведёт строка $s[l+1:k]$ (Суффикс s, имеющий длину $k-l-1$). Ещё одним правилом суфф. автомата будет то, что максимальная по длине строка, которая ведёт в B должна быть $s[l+1:k]$. И тогда все строки, которые ведут в B будут иметь вид $s[l+1:k], s[l+2:k], \ldots, s[l+t:k]$, (суффиксы s, имеющие длину от $k-l-t$ до $k-l-1$). Другими словами вершина B будет принимать следующие суффиксы s после вершины A. Тогда будем считать, что из A суффиксная ссылка ведёт в вершину B. Тогда из каждой вершины кроме стартовой будет по одной суффиксной ссылке.

Тогда заметим несколько важных вещей. Возьмём любую вершину A. Заметим, что так как все строки, которые в неё ведут, являются суффиксами друг друга, то они все заканчиваются на один и тот же символ. Значит все переходы в A ведутся по одному и тому же символу, и только по нему.

Вообще в A могут вести переходы из разных вершин. Каждой из них соответствует какая-то строка. Ту из них, которой соответствует самая длинная строка, будем называть **предком вершины** A.

Как мы уже знаем, в вершину A видут строки $s[0:k], s[1:k], s[2:k], s[3:k], \ldots, s[l:k]$. Теперь посмотрим на все вершины, из которых есть переход в A. Тогда эти вершины ведут строки $s[0:k-1], s[1:k-1], s[2:k-1], s[3:k-1], \ldots, s[l:k-1]$, и только они. И замечаем, что тогда между этими вершинами как-то расставленны суффиксные ссылки, причём если от предка A спускаться по суффиксным ссылкам, то мы будем идти по вершинам, из которых есть переход в A пока не попадём в какую-то вершину, из которой в A перехода нет. При этом мы пройдём по всем вершинам, из которых есть переход в A и в конце мы окажемся в вершине, которой соответствует строка $s[l+1:k-1]$. Заметем, что тогда из этой вершины будет идти переход суффиксную ссылку вершины A, и эта вершина как раз будет предком суффиксной ссылки A.

И наконец заметим последнюю вещь. Если из вешины есть переход по какому-то символу c, то есть вершина, в которую ведёт строка $s+c$, а значит в исходной строке был какой-то суффикс, начинающийся с $s+c$. Тогда у этой же строки есть суффикс, начинающийся с $s[l+1:k]+c$. Тогда из суффиксной ссылки вершины A тоже должен быть переход по символу c. Тогда посмотрим, куда ведут эти 2 перехода. Они могут вести в одну и ту же вершину, а могут вести в разные. Если они ведут в разные, то в первую точно ведёт строка $s[l:k]+c$ (самая минимальная строка, ведущая в A с приписаным к ней символом c), а во вторую ведёт строка $s[l+1:k]+c$ (строка, соответствующая суффиксной ссылке из A, с приписыным к ней символом c). Замечаем, что это последовательные суффиксы друг друга, значит из первой вершины должна вести суффиксная ссылка во вторую.

2.1 Алгоритм

Мы разобрались со всеми определениями, и теперь пора перейти к алгоритму. Алгоритм гораздо легче всей теории, за ним стоящей.

Будем строить суффавтомат итеративно, т.е будем достраивать суффавтомат при приписывании к исходной строке нового символа.

Посмотрим на вершину A, которой соответствует вся строка. Как мы знаем, в неё ведёт несколько последовательных суффиксов строки, а следующие суффиксы ведут в её суффиксную ссылку. Если спускаться от этой вершины по суффиксным ссылкам, то мы посетим все *терминальные* вершины суфф. автомата и только их.

Пусть мы дописали к автомату символ ch. Тогда нам надо, чтобы автомат принимал любой суффикс исходной строки с приписанным к нему символом ch. Тогда как-то менять нам надо только терминальные вершины, и то не все, ведь из никоторых уже мог быть переход по ch.

Делать мы это будем так: создадим вершину B и проведём в неё переход по символу ch из A. Далее перейдём в суффиксную ссылку из A. Назовём теперь эту вершину именем A. Если из нового A нет перехода по символу ch, то проведём его тоже в вершину B. Далее снова перейдём в суфф. ссылку из A, и снова назовём теперь эту вершину именем A. Будем продолжать так делать пока из A не появится переход по символу ch. Пусть этот переход ведёт в вершину C. У нас возможно 2 варианта: A — предок C или A — не предок вершины C.

В первом случае всё просто. Проведём суфф ссылку из B в C. Замечаем, что после этого для всех старых вершин свойства суфф автомата выполнены, в B будут вести последовательные суффиксы исходной строки с приписаным символом ch, и из B будет существовать корректная суффиксная ссылка. Тогда если в новом суффавтомате пометить терминальными все вершины, которые мы посетим спуском из B по суфф ссылкам, то автомат будет корректным и будет принимать только суффиксы начальной строки с приписаным символом ch.

Во втором случае всё тоже просто, но не совсем. Из вершины B мы не можем пустить суфф ссылку в вершину C, потому что в C ведут некотороые строки, не являющиеся суффиксами B. Но замечаем, что строка, которая соответствует A с приписанным к ней символом ch ведёт в C и является самым большим суффиксом B, не ведущим в B. Тогда сделаем следующую вешь. Возьмём вершину C и раздвоим её. Вернее отделим от неё кусок. Возьмём строку, соответствующую A, припишем к ней ch и возьмём все её суффиксы, лежащие в C. И создадим вершину D, в которую будут вести только эти строки. Чтобы эти строки вели в D, надо провести туда переходы. Замечаем, что строка, соответствующая A с приписаным ch будет соответствовать D, а значит A будет являтся предком D. Дальше будем спускаться по суффиксным ссылкам из A, и пока переход по символу ch будет вести в C, будем менять его на D, а если переход будет вести куда-то в другое место, то это будет означать, что мы пройдём все вершины, которым соответствуют суффиксы A и из которых переход по ch раньше вёл в C. После этого переходы в D будут корректно построенны. После этого построим корректно суффиксные ссылки. Проведём из C суффиксную ссылку в D и проведём из D суффиксную ссылку туда, куда она раньше вела из C. Так как вершина D является отделённым куском вершина C, то все переходы из D будут такими-же, как все переходы из C. Тогда замечаем, что после такого разделения вершины C на 2 части автомат всё ещё корректный и его правила выполняются. А так же замечаем, что теперь из A переход по ch ведёт в D, и A — предок D. Тогда мы свели этот случай к предыдущему и теперь нам достатосно построить переход из B в D, и суффавтомат построен. Разумеется, вершины надо помечать терминальными не после каждого добавления символа, а только в самом конце.

Оценим время работы. Во первых мы каждый раз создавали не более 2 вершин, т.е. всего вершин $O(n)$. Мы много раз спускались по суффиксным ссылкам из A, но назовём балансом вершины длину пути от неё до корня по суфф. ссфлкам. Тогда баланс B в первом случае это баланс C плюс 1. А баланс C не превышает баланс конечного A плюс 1, т.к. предки вершин на пути от C до корня различны и лежат на пути от конечного A до корня. Тогда если баланс B равен балансу A плюс 1 минус число

спусоков по A. Тогда как было с Ахо-карасиком, число спусков не превышает n. Про то, что раздвоение быстро работает можно тоже доказать, но мне лень это писать, так что подумайте сами. В общем там всё работает быстро и общее построение работает за $O(N)$.

2.2 Реализация

Звучит это всё страшно, но пишется в 40 строк, вот реализация на питоне с кучей ошибок, чтобы вы не копипастили.

```
# Здесь много ошибок, код нужен только для ознакомления

class node:
    def init(self):
        self.nx = [-1] * 26 # Переходы
        self.p = -1         # Суффиксная ссылка
        self.pre = -1       # Предок

s = [node()]

# Мы по умолчанию считаем, что суффиксная ссылка из корня ведёт в -1

def add(a, ch): # a - последняя вершина в автомате, ch - добавляемый символ
    # Создаём вершину b и указываем её правильные параметры
    b = len(s)
    s.append(node())
    s[b].p = 0 # Обратите внимание на это, возможно мы так и не дойдём до вершины C
               # и тогда суффиксная ссылка из b должна указывать на 0
    s[b].pre = -1
    while a != -1:
        if (s[a].nx[ch] == -1):
            # Случай, когда из a нет перехода по ch
            s[b].nx[ch] = a
            a = s[a].p
            continue
        # c - то, куда ведёт переход из a по ch
        c = s[a].nx[ch]
        # Проверяем, что предок c это a и в этом случае меням суфф. ссылку b
        if (s[c].pre == a):
            s[b].p = a
            break
        # Иначе создаём вершину d
        d = len(s)
        s.append(node())
        # Меняем суфф. ссылки
        s[c].p = d
        s[b].p = d
        s[d].p = s[c].p
        # Назначаем предка d
        s[d].pre = b
        # Копируем переходы
        s[d].nx = s[c].nx
        # Добавляем предков вершины d, пройдсь вниз по суфф. ссылкам из a
        while (a != -1 and s[a].nx[ch] == b):
            s[a].nx[ch] = d
            a = s[a].pre
        break
    # Если строк несколько, то возможно мы не добавили b. Поэтому надо возвращать вот это
    return s[s[b].pre].nx[ch]
```

3 Задачи

3.1 Количество различных подстрок

Замечаем, что любая подстрока строки — это начало какого-то суффикса. Тогда в нашем автомате любой путь — это подстрока начальной строки, и для любой подстроки существует путь в автомате по ней. Тогда первый способ найти число подстрок данной строки — посчитать число путей в автомате. Это можно сделать обычным ДФСом, путём посчёта для каждой вершины числа путей из неё (только надо запоминать уже посещённые вершины, чтобы 2 раза не ходить). Есть второй способ. Каждый путь заканчивается в какой-то вершине. Тогда для каждой вершины можно посчитать число путей, в неё ведущих. Это можно делать как в процессе построения (но не забывать про пересчёт числа путей в раздваиваемые вершины), так и после. Тогда сумма по кол-ву путей, заканчивающихся в каждой вершине автомата это число различных путей в автомате, т.е. число подстрок данной строки. Этот способ так же позволяет найти число подстрок у каждого префикса строки, так как в процессе построения автомата мы по очерёдно строим суфф. автомат для каждого префикса нашей строки.

3.2 Рефрен

Рефрен это такая подстрока данной нам строки, что произведение её длины на число вхождений в нашу строку максимально. Найти рефрен очень просто. Замечаем, что число вхождений подстроки в строку это число суффиксов строки, что эта подстрока — начало этого суффикса. Тогда число вхождений подстроки в строку это число путей из вершины, соответствующей этой подстроке, в терминальные вершины. Тогда для каждой вершины посчитаем число путей из неё в терминальные вершины (это делается так же, как и подсчёт числа просто путей из вершины), а так же для каждой вершины посчитаем длину строки, которая ей соответствует. Переберём все вершины и найдём ту, у которой произведение длины на число путей до терминальных вершин будет максимально. Строка, соответствующая этой вершине это и есть рефрен.

3.3 Важное замечание

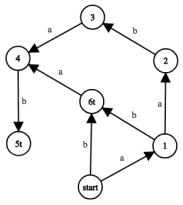
Пока мы строили суфф. автомат для одной строки. Но замечаем, что при добавлении символа мы никак не использовали то, что суфф. автомат должен быть конкретно для одной строки. Мы использовали только то, что вершина принимает последовательные суффиксы и то, что существует суффиксная ссылка. В частности это означало, что если в суфф. автомате есть путь, соответствующий какой-то строке, то есть путь, соответствующий любому суффиксу данной строки, а значит и вообще луюбой подстроке данной строки. Тогда заметим, что мы можем просто взять любую вершину, и дописать к ней переход по какому-то символу, используя наш алгоритм. И накаких проблем от этого не будет. Тогда перейдём к следующей задаче.

3.4 Подстрока в k строках из набора

Дан набор из n строк, с суммарной длиной m. Для каждого $1 \leq k \leq n$ найти максимальную длину строки, являющейся подстрокой хотя-бы k строк из набора.

Для начала построим суфф. автомат, который принимает все суффиксы всех строк из набора. Это делается так: сначала построим автомат для первой строки. Дальше пользуясь важным замечанием возьмём корень автомата, и припишем к нему первый символ второй строки. К этой вершине припишем второй символ второй строки и т. д. (Обратите внимание, что функцию *add* надо вызывать даже если переход по символу уже был, так как возможно нам придётся раздваивать следующую вершину. Именно поэтому в функции *add* из примера такой странный return, ведь возможно мы не добавляли перехода в вершину b). Так мы получим автомат, принимающий все суффиксы первых двух строк. Дальше так же туда припишем третью, червёртую и т.д. строки. По сути у нас автомат превратился в бор с уже готовыми суфф. ссылками и принимающий не только строки, но и все их подстроки.

Вы наверное думаете, что всякие-там термины уже закончились. А вот нет. **Подавтоматом вершины** A называется такая часть вершин нашего автомата, что в ней есть пути по всем подстрокам строки, соответствующей вершине A, и только по ним. Если рассмотреть уже встречавшийся ранее пример суфф. автомата для строки «aabab», то в подавтомат вершины 3 войдут вершины $start, 1, 2, 3, 6t$, в подавтомат вершины $6t$ войдут вершины $start, 1, 6t$, а в подавтомат вершины 2 войдут только вершины $start, 1, 2$.



Заметим, что чтобы построить подватомат вершины A, нам достаточно построить подавтомат её предка, а так же добавить туда все вершины, достижимые из A спуском по суфф. ссылкам. (Это верно, так как подавтомат предка принимает подстроки A без последнего символа, а оставшиеся подстроки A это суффиксы A, т.е. то, что мы пройдём, когда будем спускаться по суфф. ссылкам. Заметим, что тогда мы можем сказать, что подавтомат вершины A это сама вершина A и объединение подавтоматов её суфф. ссылки и её предка. А тогда с помощью очень простой рекурсивной функции *subautomaton* мы можем пройти по всем вершинам подавтомата.

```
class node:
    def init(self):
        self.nx = [-1] * 26 # Переходы
        self.p = -1         # Суффиксная ссылка
        self.pre = -1       # Предок
        self.last_used = -1 # Время последнего посещения

s = [node()] # Список вершин

t = 0 # Временная метка построения подавтомата
      # (нужно чтобы не посещать одну и ту же вершину несколько раз)

# Делает какое-то действие для подавтомата вершины
# При каждом следующем запуске временную метку t надо менять
def subautomaton(a):
    # Проверка, что при этом построении подавтомата мы ещё не посещали вершину
    if (s[a].last_used == t):
        return
    s[a].last_used = t
    # Делаем какое-то действие
    subautomaton(s[a].p) # Идём в суфф. ссылку
    subautomaton(s[a].pre) # Идём в предка
```

Теперь вернёмся к задаче. Попробуем для каждой строки найти число строк из набора, а которых эта строка является подстрокой. Для этого нам надо для каждой вершины узнать, у скольких строк из из набора эта вершина является подстрокой. А для этого нам достаточно посмотреть на каждую вершину, соответствующую какой-то строке из набора. Мы знаем, что все вершины в её подавтомате как раз являются её подстроками. Тогда для каждой вершины заведём счётчик числа строк, в которой она является подавтоматом. И для каждой строки из набора прибавим 1 к счётчикам всех строк в подавтомате. Для каждой вершины так же запомним длину строки, соответствующей этой вершине. И дальше просто переберём все вершины и обновим через них ответ.

Остаётся один непонятный вопрос — время работы нашего алгоритма. Это только в обычном суффавтомате число вершин $O(n)$. Но тут берётся подавтомат, и вообще может оказаться, что каждой подстроке подавтомата соответствует по отдельной вершине, и тогда у нас будет l^2 вершин в подавтомате, где l — длина строки, для которой мы строим подавтомат. Что-же, получается наш алгоритм работает за $O(n \cdot m^2)$. А вот нифига подобного. У нас всего вершин в автомате будет $O(m)$, значит число вершин в податомате не больше $O(m)$. Но асимптотика $O(n \cdot m)$ нас тоже не очень-то устраивает. Тогда разделим все строки из набра на короткие и длинные. В коротких число символов должно быть не больше $\sqrt{m}$, а в длинных может быть больше. Заметим, что число длинных строк не больше $\sqrt{m}$, значит величина всех их подавтоматов $O(m \cdot \sqrt{m})$. А что делать с короткими строками. Как мы знаем, для строки l, размер её подавтомата не больше l^2. А это меньше, чем $l \cdot \sqrt{m}$. Тогда суммарный размер подавтоматов коротких строк это $\sum l \cdot \sqrt{m} = \sqrt{m} \cdot \sum l$. А сумма длин всех строк не больше m. Тогда суммарный размер всех подавтоматов $O(m\sqrt{m})$. Но так как про эту идею никто не знает, никто не строит антитесты, на которых это работает за $O(m\sqrt{m})$. Поэтому этот алгоритм просто летает и спокойно заходит даже при $m = 5\,000\,000$. И вот это вот позволяет вам становиться богом строк и решать почти любую задачу на одну или несколько строк. Скажем вот:

3.5 Ахо-карасик в онлайне с линейной памятью

Есть 3 онлайн запроса: добавить в набор строку, удалить из набора строку и для данной строки узнать суммарное число вхождений строк из набора в качестве подстроки.

Запросы первого и второго вида будем обрабатывать очень просто. Если пришла новая строка, то добавим её в автомат и пометим её последнюю вершину терминальной. Если пришёл запрос удаления, то уберём терминальную пометку.

На запрос третьего типа будем отвечать так: добавим строку в автомат, пройдёмся по вершинам, отвечающим за префиксы этой строки. И будем спускаться из них по суфф. ссылкам и если прошли через терминальную вершину, то прибавлять 1 к ответу. Логично, что это будет долго работать. Но сделаем так: если мы оказались в какой-то вершине, через которую уже так вот спускались по суфф. ссылкам, то не будем идти так же ещё раз, а просто прибавим заранее посчитанный ответ для неё. Тогда по каждой вершине подавтомата данной строки мы пройдёмся по разу. А значит время работы будет $O(m \cdot \sqrt{m})$.

4 Завершение

Суфф. автоматом можно решать много других задач. Код суфф. автомата очень простой, и часто для построения даже обычного ахо карасика можно его заюзать. Так же если вдруг вам припрёт написать задачу не суфф. автоматом, а суфф. деревом или суфф. массом, то сообщаем, что Укконен — это явно не то, чем стоит строить суфф. дерево (а из него уже суфф. масс). Просто постройте суфф. автомат для перевёрнутой строки, возьмите дерево суффиксных ссылок, это и будет суфф. дерево для начальной строки. Доказательство этого факта оставим читателю в качестве упражнения.

И ещё раз напомню, что для дебага суфф. автомата надо использовать строку «aabab». Если суфф. автомат на ней правильно построен, то с вероятностью 99% ваш код правильный.