Суффиксные автоматы 2018

Предварительные сведения

Позиции окончаний endpos**,**

- Рассмотрим любую непустую подстроку t строки s. Тогда назовём **множеством окончаний** endpos(t) множество всех позиций в строке s, в которых оканчиваются вхождения строки t.

endpos –эквивалентные строки

- Назовем две подстроки t_1 и t_2 endpos - эквивалентными, если их множества окончаний совпадают: endpos (t_1) = endpos(t_2).
- Таким образом, все непустые подстроки строки s можно разбить на несколько **классов эквивалентности** соответственно их множествам endpos.

endpos –эквивалентные строки

- В суффиксном автомате - **эквивалентным подстрокам соответствует одно и то же состояние.**
- Число состояний в суффиксном автомате равно количеству классов endpos-эквивалентности среди всех подстрок, плюс одно начальное состояние.

endpos –эквивалентные строки

- Каждому состоянию суффиксного автомата соответствуют одна или несколько подстрок, имеющих одно и то же значение .
- **Примем как аксиому**, и опишем алгоритм построения суффиксного автомата, исходя из этого предположения

endpos –эквивалентные строки. Дополнительные замечания

- **Лемма 1**. Две непустые подстроки u и w (length(u)≤ length(w)) являются endpos -эквивалентными тогда и только тогда, когда строка u встречается в строке s только в виде суффикса строки w .

Дополнительные замечания

- **Лемма 2**. Рассмотрим две непустые подстроки u и w ((length(u)≤ length(w))). Тогда их множества endpos либо не пересекаются, либо endpos(w) целиком содержится в endpos (u) , причём это зависит от того, является u суффиксом w или нет.

Доказательство леммы 2

- Пусть множества endpos(u) и endpos (w) имеют хотя бы один общий элемент. Это означает, что строки u и w оканчиваются в одном и том же месте, т. е. u— суффикс w. Но тогда каждое вхождение строки w содержит на своём конце вхождение строки u, что и означает, что его множество endpos (w) целиком вкладывается в множество endpos (u) .

Дополнительные замечания

- **Лемма 3**. Рассмотрим некоторый класс endpos -эквивалентности. Отсортируем все подстроки, входящие в этот класс, по невозрастанию длины. Тогда в получившейся последовательности каждая подстрока будет на единицу короче предыдущей, и при этом являться суффиксом предыдущей. Иными словами, **подстроки, входящие в один класс эквивалентности, на самом деле являются суффиксами друг друга, и принимают всевозможные различные длины на некотором отрезке [x,y]** .

Доказательство леммы 3(1)

- Зафиксируем некоторый класс endpos -эквивалентности. Если он содержит только одну строку, то корректность леммы очевидна. Пусть теперь количество строк больше одной.
- Согласно лемме 1, две различные endpos -эквивалентные строки всегда таковы, что одна является собственным суффиксом другой. Следовательно, в одном классе endpos -эквивалентности не может быть строк одинаковой длины.

Доказательство леммы 3(2)

- Обозначим через w длиннейшую, а через u— кратчайшую строку в данном классе эквивалентности. Согласно лемме 1, строка u является собственным суффиксом строки w . Рассмотрим теперь любой суффикс строки w с длиной в отрезке [length(u); length(w)], и покажем, что он содержится в этом же классе эквивалентности. В самом деле, этот суффикс может входить в s только в виде суффикса строки w (поскольку более короткий суффикс u входит только в виде суффикса строки w). Следовательно, согласно лемме 1, этот суффикс endpos -эквивалентен строке w, что и требовалось доказать.

Суффиксные ссылки

- Рассмотрим некоторое состояние автомата $v \neq t_0$. Состоянию v соответствует некоторый класс строк с одинаковыми значениями endpos, причём если мы обозначим через w длиннейшую из этих строк, то все остальные будут суффиксами w.
- Также мы знаем, что первые несколько суффиксов строки w (если мы рассматриваем суффиксы в порядке убывания их длины) содержатся в том же самом классе эквивалентности, а все остальные суффиксы (как минимум, пустой суффикс) — в каких-то других классах. Обозначим через t первый такой суффикс — в него мы и проведём суффиксную ссылку.

Суффиксные ссылки

- Иными словами, **суффиксная ссылка** link(v) ведёт в такое состояние, которому соответствует **наидлиннейший суффикс** строки w, находящийся в другом классе endpos-эквивалентности.
- Считаем, что начальному состоянию t_0 соответствует отдельный класс эквивалентности (содержащий только пустую строку), и полагаем endpos(t_0)=[-1…length(s)-1].

Суффиксные ссылки

- **Лемма 4**. Суффиксные ссылки образуют **дерево**, корнем которого является начальное состояние t_0 .
- Доказательство. Рассмотрим произвольное состояние $v \neq t_0$. Суффиксная ссылка link(v) ведёт из него в состояние, которому соответствуют строки строго меньшей длины (это следует из определения суффиксной ссылки и из леммы 3). Следовательно, двигаясь по суффиксным ссылкам, мы рано или поздно придём из состояния v в начальное состояние t_0 , которому соответствует пустая строка.

Суффиксные ссылки

Лемма 5. Построим из всех имеющихся множеств endpos **дерево** (по принципу "множество-родитель содержит как подмножества всех своих детей"), то оно будет совпадать по структуре с деревом суффиксных ссылок.

Доказательство.

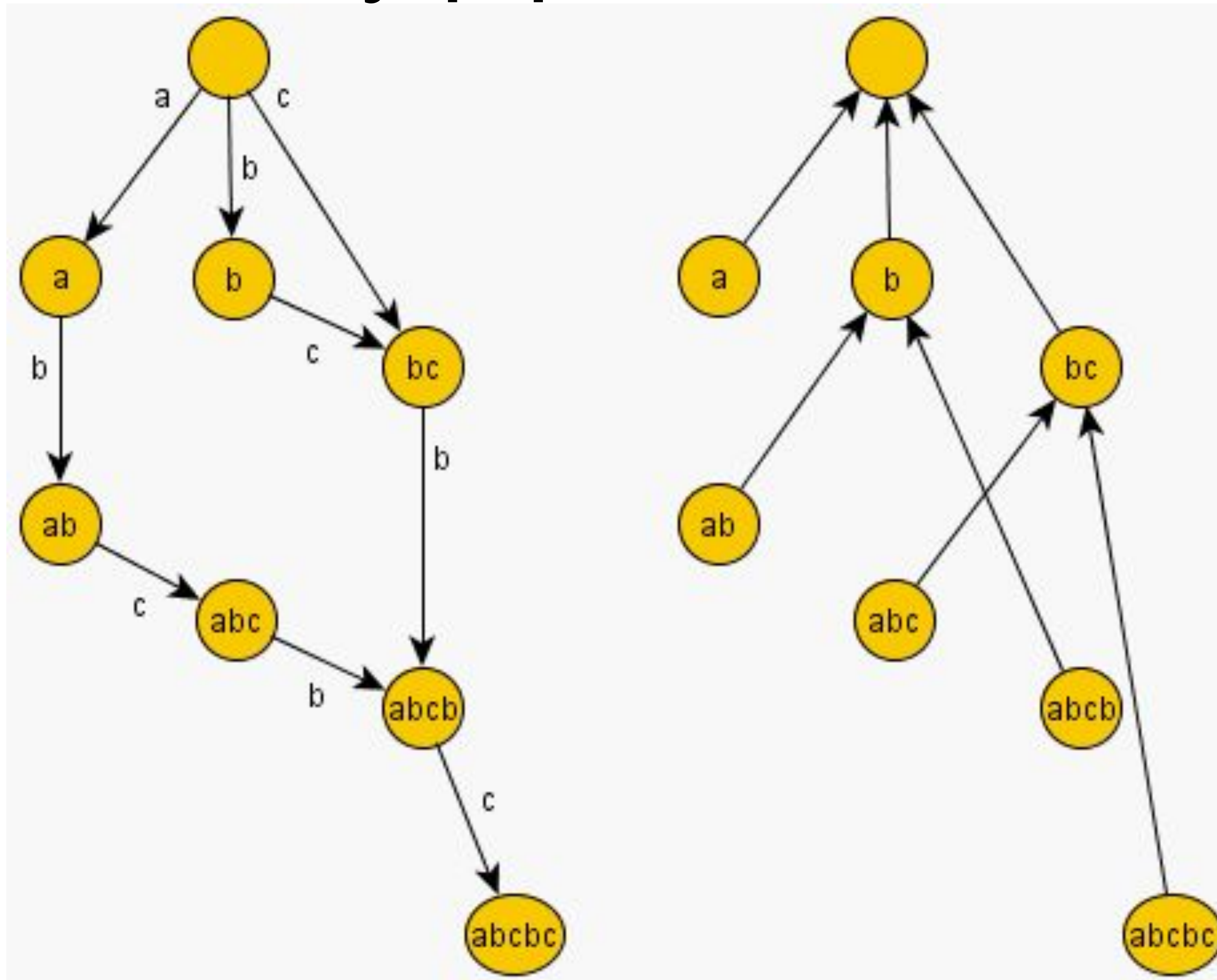
То, что из множеств endpos можно построить дерево, следует из леммы 2 (о том, что любые два множества endpos либо не пересекаются, либо одно содержится в другом).

Лемма 5(продолжение)

Рассмотрим теперь произвольное состояние $v \neq t_0$ и его суффиксную ссылку link(v). Из определения суффиксной ссылки и из леммы 2 следует: $endpos(v) \subset endpos(link(v))$.

- Что с предыдущей леммой и доказывает утверждение: дерево суффиксных ссылок по сути есть дерево вкладывающихся множеств endpos.

Пример дерева суффиксных ссылок в суффиксном автомате



Предварительные итоги-1

- Множество подстрок строки s можно разбить на классы эквивалентности согласно их множествам окончания endpos.
- Суффиксный автомат состоит из начального состояния t_0 , а также по одному состоянию на каждый класс endpos -эквивалентности.
- Каждому состоянию v соответствует одна или несколько строк. Обозначим через longest(v) длиннейшую из таких строк, через len(v) её длину. Обозначим через shortest(v) кратчайшую из таких строк, а её длину через minlen(v) .

Предварительные итоги-2

- Тогда все строки, соответствующие этому состоянию, являются различными суффиксами строки longest(v) и имеют всевозможные длины в отрезке [minlen(v);len(v)] .
- Для каждого состояния $v \neq t_0$ определена суффиксная ссылка, ведущая в такое состояние, которое соответствует суффиксу строки longest(v) длины minlen(v)-1. Суффиксные ссылки образуют дерево с корнем в t_0 , причём это дерево, по сути, является деревом отношений включения между множествами endpos.

Предварительные итоги-3

Таким образом, $\text{minlen}(v)$ для $v \neq t_0$ выражается с помощью суффиксной ссылки $\text{link}(v)$ как:

$\text{minlen}(v) \equiv \text{len}(\text{link}(v))+1$.

Если мы стартуем из произвольного состояния v_i и будем идти по суффиксным ссылкам, то рано или поздно дойдём до начального состояния t_0.

При этом получится последовательность непересекающихся отрезков $[\text{minlen}(v_i);\text{len}(v_i)]$, которые в объединении дадут один сплошной отрезок.

Алгоритм построения суффиксного автомата за линейное время

- Алгоритм **онлайновый**, т.е. будет добавлять по одному символу строки s , перестраивая соответствующим образом текущий автомат.
- Чтобы расход памяти был линейным, в каждом состоянии будем хранить только значение len , link и список переходов из этого состояния. Метки терминальных состояний не поддерживается (покажем, как расставить эти метки после построения суффиксного автомата, если в них есть необходимость).

Предварительные сведения

- **Изначально** автомат состоит из единственного состояния t_0 , которое условимся считать нулевым состоянием (остальные состояния будут получать номера 1,2,…). Присвоим этому состоянию len=0 , а значению link присвоим для удобства -1 (означающее ссылку на фиктивное, несуществующее состояние).

Рассмотрим реализацию обработки добавления 1 символа в конец текущей строки

- Пусть last— это состояние, соответствующее всей текущей строке до добавления символа c. (Изначально last=0, а после добавления каждого символа мы будем менять значение last)
- Создадим новое состояние cur, проставив ему len(cur)=len(last)+1. Значение link(cur) пока считаем неопределённым.

Сделаем такой цикл: изначально мы стоим в состоянии last; если из него нет перехода по букве c, то добавляем этот переход по букве c в состояние cur, и затем переходим по суффиксной ссылке, снова проверяя — если нет перехода, то добавляем. Если в какой-то момент случится, что такой переход уже есть, то останавливаемся — и обозначим через p номер состояния, в котором это произошло.

- Если ни разу не случилось, что переход по букве c уже имелся, и мы так и дошли до фиктивного состояния -1(в которое мы попали по суффиксной ссылке из начального состояния t_0), то мы можем просто присвоить $link(cur)=0$ и выйти.
- Допустим теперь, что мы остановились на некотором состоянии p, из которого уже был переход по букве c. Обозначим через q то состояние, куда ведёт этот имеющийся переход.

Рассмотрим 2 случая

- Разделяются по выполнению или нет условия len(p)+1 = len(q).
- Если len(p)+1 = len(q) , то можно просто присвоить link(cur)=q и выйти.
- В противном случае, всё несколько сложнее. Необходимо произвести "**клонирование**" состояния q : создать новое состояние clone, скопировав в него все данные из вершины q(суффиксную ссылку, переходы), за исключением значения len : надо присвоить len (clone) = len(p)+1.

- После клонирования мы проводим суффиксную ссылку из cur в это состояние clone, также перенаправляем суффиксную ссылку из q в clone .
- И, последнее, что мы должны сделать — это пройтись от состояния p по суффиксным ссылкам, и для каждого очередного состояния проверять: если имелся переход по букве c в состояние q , то перенаправлять его в состояние clone

(а если нет, то останавливаться).

И, чем бы ни закончилось выполнение этой процедуры, в конце обновляем значение last , присваивая ему cur .

Терминальные вершины

- Если нужно знать, какие вершины являются **терминальными**, а какие — нет, то можно найти все терминальные вершины после построения суффиксного автомата для всей строки. Для этого рассмотрим состояние, соответствующее всей строке (оно, очевидно, сохранено в переменной last), и проходим по его суффиксным ссылкам, пока не дойдём до начального состояния, помечая каждое пройденное состояние как терминальное. Легко понять, что тем самым мы пометим состояния, соответствующие всем суффиксам строки s , что и требовалось.

Замечания

- Отметим, что добавление одного символа приводит к добавлению одного или двух состояний в автомат. Таким образом, **линейность числа состояний** очевидна.
- Линейность числа переходов, да и линейное время работы алгоритма будут доказаны после доказательства корректности алгоритма.

Доказательство корректности алгоритма

- Назовём переход (p,q) **сплошным**, если len(p)+1 = len(q). В противном случае, т.е. когда len(p)+1 < len(q), переход будем называть **несплошным.**
- Из описания алгоритма можно увидеть, что сплошные и несплошные переходы приводят к разным ветвям алгоритма. Сплошные переходы называются так потому, что, появившись впервые, они больше никогда не будут меняться. В противоположность им, несплошные переходы могут измениться при добавлении новых букв к строке (измениться может конец дуги-перехода).

Доказательство корректности алгоритма-пр1

- Во избежание неоднозначностей, под строкой s мы будем подразумевать строку, для которой был построен суффиксный автомат до добавления текущего символа c.
- Алгоритм начинается с того, что мы создаём новое состояние cur , которому будет соответствовать вся строка s+c. Понятно, почему мы обязаны создать новое состояние — т.к. вместе с добавлением нового символа возникает новый класс эквивалентности — это класс строк, оканчивающихся на добавляемом символе c.

Доказательство корректности алгоритма-пр2

- После создания нового состояния алгоритм проходится по суффиксным ссылкам, начиная с состояния, соответствующего всей строке s , и пытается добавить переход по символу c в состояние cur. Тем самым, приписывая к каждому суффиксу строки s символ c. Но добавлять новые переходы можно только в том случае, если они не будут конфликтовать с уже имеющимися, поэтому, как только встречается уже имеющийся переход по символу c, сразу же необходимо остановиться.

Доказательство корректности алгоритма-пр3

- Самый простой случай — если так и доходим до фиктивного состояния -1 , добавив везде по новому переходу вдоль символа c. Это означает, что символ c в строке ранее не встречался. Успешно добавляем все переходы, осталось только проставить суффиксную ссылку у состояния cur — она, очевидно, должна быть равна 0 , поскольку состоянию cur в данном случае соответствуют все суффиксы строки s+c.

Доказательство корректности алгоритма-пр4

- Второй случай — когда мы наткнулись на уже имеющийся переход (p,q) . Это означает, что мы пытались добавить в автомат строку x+c (где x — некоторый суффикс строки s , имеющий длину len(p)), а эта строка **уже была ранее добавлена** в автомат (т.е. строка x+c уже входит как подстрока в строку s). Поскольку мы предполагаем, что автомат для строки s построен корректно, то новых переходов мы больше добавлять не должны.

Доказательство корректности алгоритма-пр5

- Возникает сложность с тем, куда вести суффиксную ссылку из состояния cur.

  Требуется провести суффиксную ссылку в такое состояние, в котором длиннейшей строкой будет являться как раз эта самая x+c, т.е. len для этого состояния должен быть равен len(p)+1. Однако такого состояния могло и не существовать: в таком случае нам надо произвести **"расщепление"** состояния.

Доказательство корректности алгоритма-пр6

- Итак, по одному из возможных сценариев, переход (p,q) оказался сплошным, т.е. len(q)=len(p)+1 . В этом случае всё просто, никакого расщепления производить не надо, и мы просто проводим суффиксную ссылку из состояния cur в состояние q.

Доказательство корректности алгоритма-пр7

- Более сложный вариант — когда переход несплошной, т.е. len(q)>len(p)+1 . Это означает, что состоянию q соответствует не только нужная нам подстрока w+c длины len(p)+1 , но также и подстроки большей длины. Необходимо произвести **"расщепление"** состояния q: разбить отрезок строк, соответствующих ей, на два подотрезка, так что первый будет заканчиваться как раз длиной len(p)+1 .

Доказательство корректности алгоритма-пр8

- Как производить это расщепление? **“Клонируем"** состояние q , делая его копию clone с параметром len(clone)=len(p)+1 . Копируем в clone из q все переходы, поскольку не хотим менять пути, проходившие через q . Суффиксную ссылку из clone ведём туда, куда вела старая суффиксная ссылка из q, а ссылку из q направляем в clone.
- После клонирования проводим суффиксную ссылку из cur в clone — то, ради чего мы и производили клонирование.

Доказательство корректности алгоритма-пр9

- Остался последний шаг — перенаправить некоторые входящие в q переходы, перенаправив их на clone . Какие именно входящие переходы надо перенаправить? Достаточно перенаправить только переходы, соответствующие всем суффиксам строки w+c , т.е. надо продолжить двигаться по суффиксным ссылкам, начиная с вершины p , и до тех пор, пока мы не дойдём до фиктивного состояния -1 или не дойдём до состояния, переход из которого ведёт в состояние, отличное от q .

Доказательство линейного числа операций

- Список переходов из одной вершины надо хранить в виде сбалансированного дерева, позволяющего быстро производить операции поиска по ключу и добавления ключа. Следовательно, если мы обозначим через k размер алфавита, то асимптотика алгоритма составит $O(n \log k)$ при $O(n)$ памяти.

Доказательство линейного числа операций-пр1

- Операции поиска перехода по символу, добавления перехода, поиск следующего перехода —считаются работающими за O(1).

Доказательство линейного числа операций-пр2

- Если мы рассмотрим все части алгоритма, то он содержит три части, линейная асимптотика которых не очевидна:
- Первая часть— это проход по суффиксным ссылкам от состояния last с добавлением рёбер по символу c .
- Вторая часть— копирование переходов при клонировании состояния q в новое состояние clone .
- Третья часть— перенаправление переходов, ведущих в q , на clone .

Доказательство линейного числа операций-пр3

- Воспользуемся известным фактом, что размер суффиксного автомата (как по числу состояний, так и по числу переходов) **линеен**. (Доказательством линейности по числу состояний является сам алгоритм, а доказательство линейности по числу переходов мы приведём ниже, после реализации алгоритма.).

Доказательство линейного числа операций-пр4

- Тогда очевидна линейная суммарная асимптотика **первой и второй частей**: ведь каждая операция здесь добавляет в автомат один новый переход.
- Осталось оценить суммарную асимптотику **в третьей части** — в той, где перенаправляются переходы, ведущие в q , на clone.

Доказательство линейного числа операций-пр5

- Обозначим v=longest(p). Это суффикс строки s , и с каждой итерацией его длина убывает — а, значит, и позиция v как суффикса строки s монотонно возрастает с каждой итерацией. При этом, если перед первой итерацией цикла соответствующая строка v была на глубине k (k≥2) от last(если считать глубиной число суффиксных ссылок, которые надо пройти), то после последней итерации строка v+c станет 2 -ой суффиксной ссылкой на пути от cur (которое станет новым значением last).

Доказательство линейного числа операций-пр6

- Таким образом, каждая итерация этого цикла приводит к тому, что позиция строки longest(link(link(last))), как суффикса всей текущей строки будет монотонно увеличиваться. Следовательно, всего этот цикл не мог отработать более n итераций, **что и требовалось доказать**.
- (Стоит заметить, что аналогичные аргументы можно использовать и для доказательства линейности работы первой части, вместо ссылки на доказательство линейности числа состояний.)

Реализация алгоритма

- Опишем структуру данных, которая будет хранить всю информацию о конкретном переходе (len ,link , список переходов). При необходимости можно добавить другую требуемую информацию. Список переходов мы храним в виде стандартного контейнера map, что позволяет достичь суммарно O(n) памяти и O(n log k) времени на обработку всей строки.
- struct state {
- int len, link;
- map<char,int> next;
- };