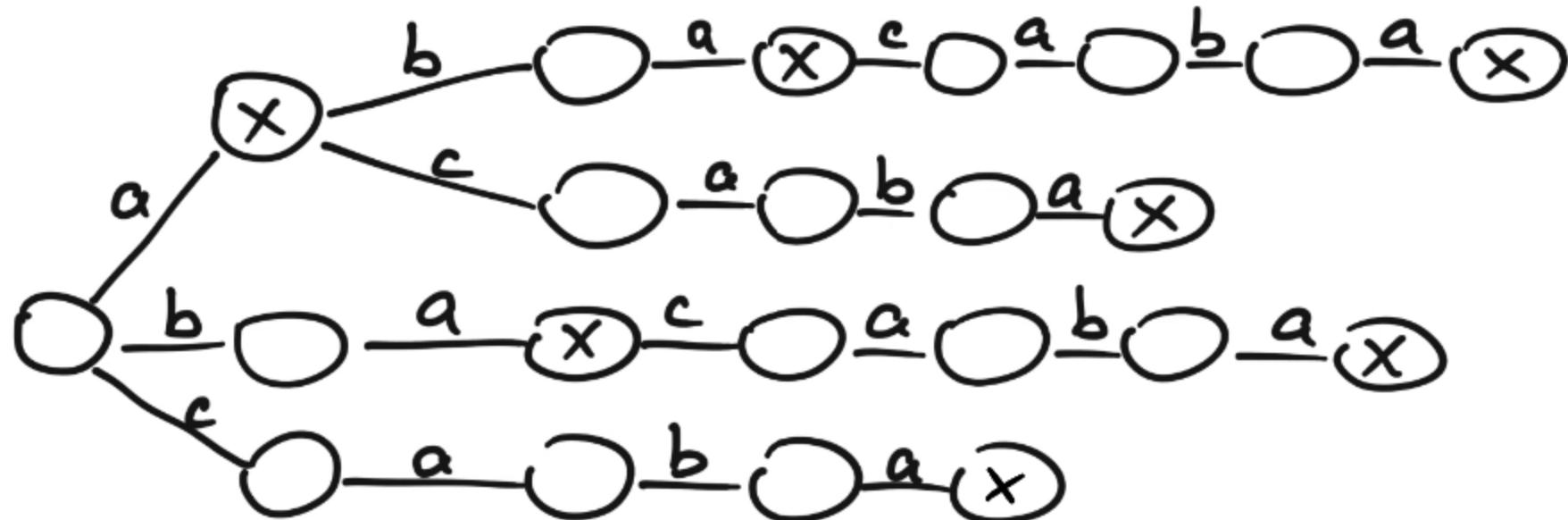


Суффиксный автомат

Суффиксный бор

Суффиксный бор строки S - бор, содержащий все суффиксы строки S .

$$S = abacaba$$

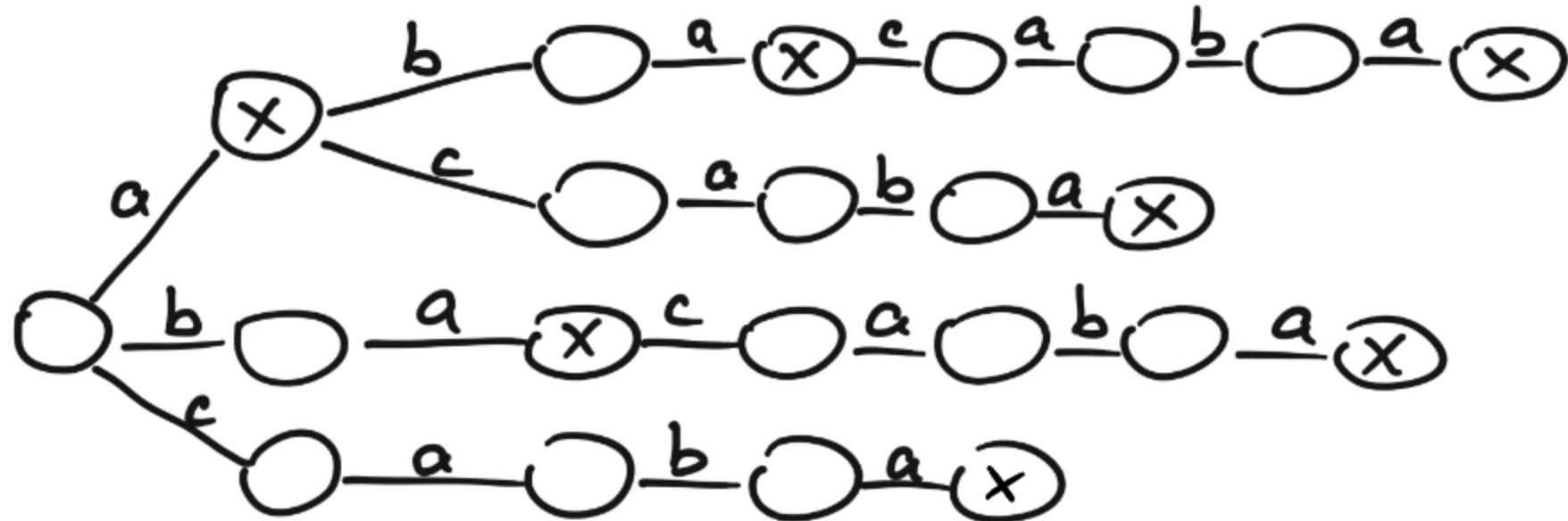


- В чем польза полученной структуры?

Суффиксный бор

Суффиксный бор строки S - бор, содержащий все суффиксы строки S .

$S = abacaba$

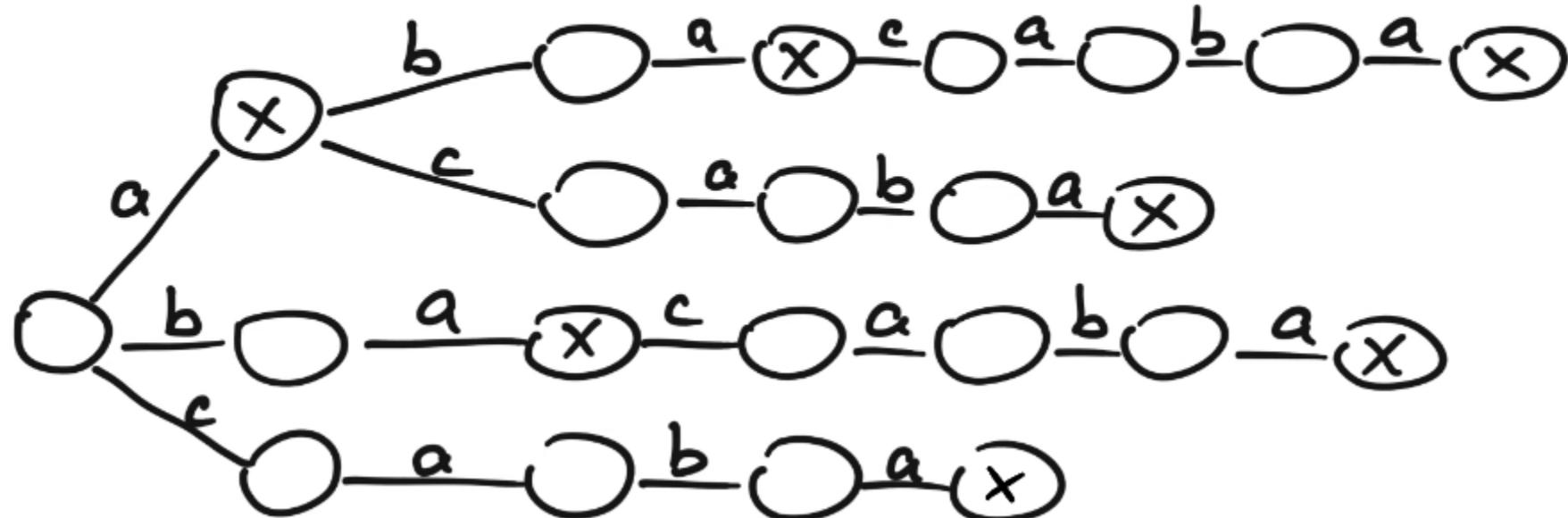


- Подстрока = префикс суффикса. \Rightarrow Каждая вершина соответствует некоторой подстроке. \Rightarrow Эффективный поиск подстроки.

Суффиксный бор

Суффиксный бор строки S - бор, содержащий все суффиксы строки S .

$$S = abacaba$$

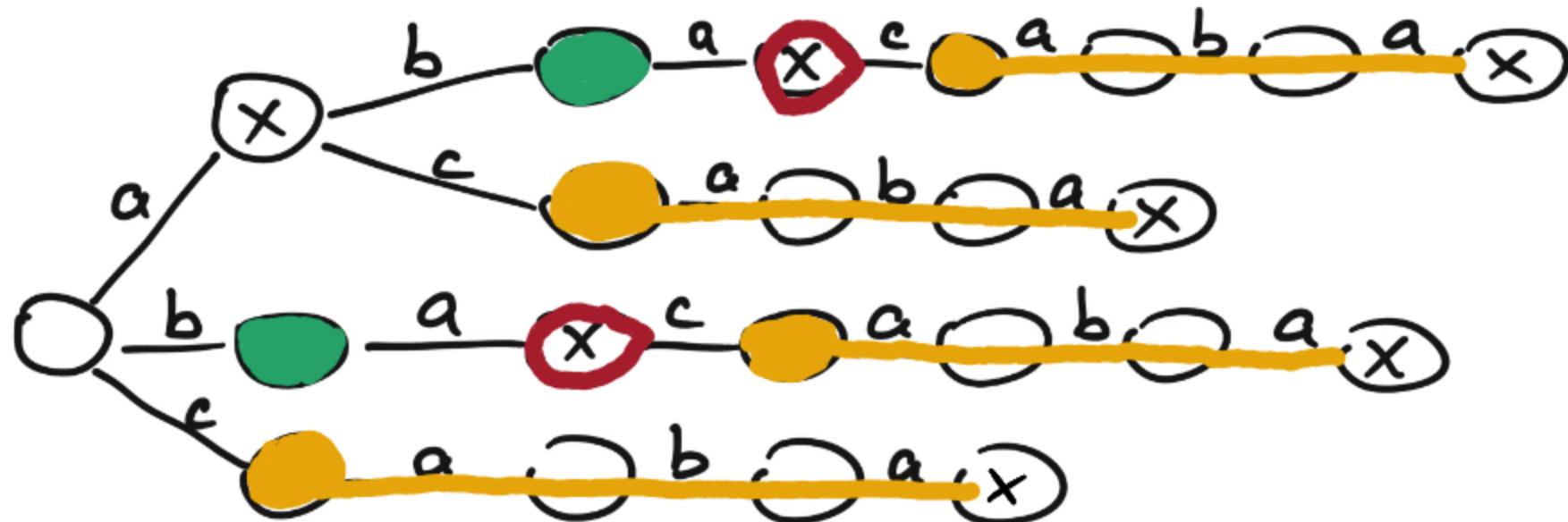


- Проблема: количество вершин $O(\sum P_i) = O(n^2)$

Суффиксный бор

Суффиксный бор строки S - бор, содержащий все суффиксы строки S .

$S = abacaba$

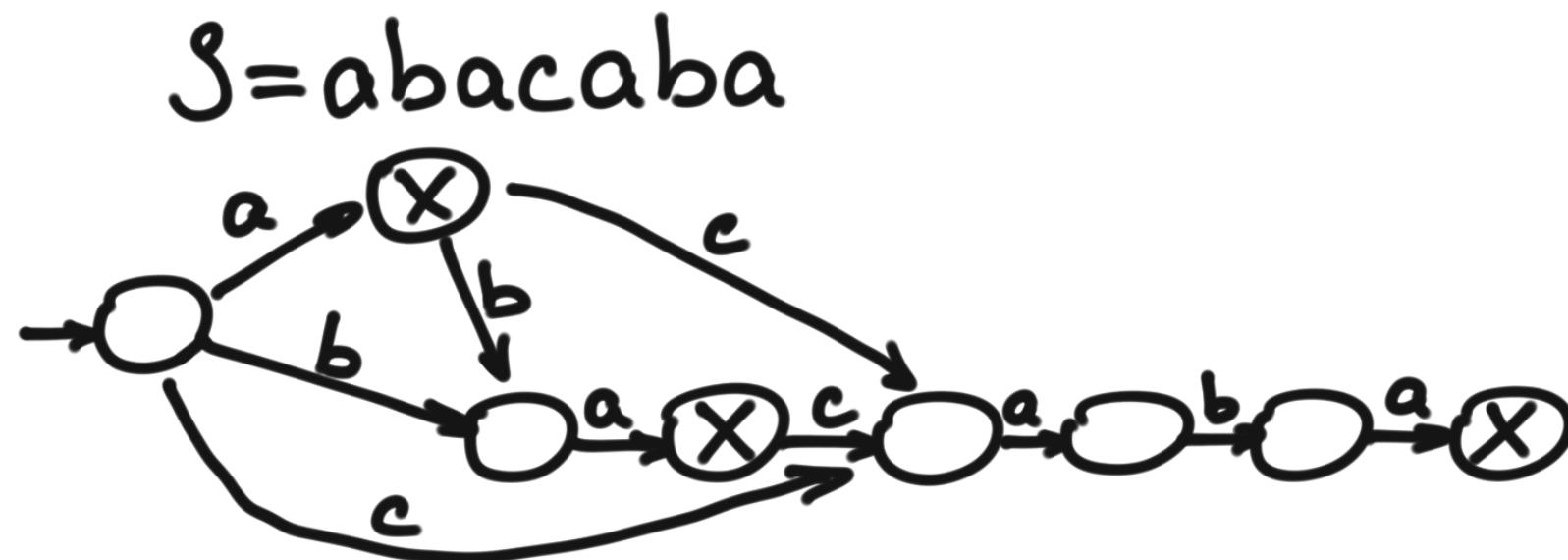


- Заметим, что в боре много повторяющихся вершин - тех, у которых совпадают поддеревья.

Суффиксный автомат

Суффиксный автомат строки S - минимальный ДКА (детерминированный конечный автомат), принимающий все суффиксы строки S .

Короче, берем суффиксный бор, склеиваем все одинаковые вершины - получаем суффиксный автомат.



Замечание: автомат не обязательно является деревом!

Суффиксный автомат

Короче, берем суффиксный бор, склеиваем все одинаковые вершины - получаем суффиксный автомат.

Кажется, из этой фразы можно сформировать алгоритм построения суффиксного автомата.



Суффиксный автомат

Короче, берем суффиксный бор, склеиваем все одинаковые вершины - получаем суффиксный автомат.

Кажется, из этой фразы можно сформировать алгоритм построения суффиксного автомата.

Проблема в том, что для построения исходного бора придется потратить $O(n^2)$ времени.

Математические основы суффиксных автоматов

Правые контексты

Обозначение: ε - пустая строка.

Опр. Правым контекстом подстроки u строки S будем называть множество $R_S(u) = \{v: uv - \text{суффикс строки } S\}$.

То есть правый контекст - множество возможных продолжений до полноценного суффикса.

Пример: $S = abacaba$

- $R_S(ab) = \{acaba, a\}$
- $R_S(a) = \{bacaba, caba, ba, \varepsilon\}$
- $R_S(ca) = \{ba\}$

Правые контексты

Опр. Назовем подстроки u и v строки S эквивалентными по правому контексту, если $R_S(u) = R_S(v)$ (их можно продолжить одинаковыми подстроками).

Пример: $S = abacaba$

- $R_S(b) = R_S(ab) = \{acaba, a\}$
- $R_S(ba) = R_S(aba) = \{caba, \varepsilon\}$
- $R_S(c) = R_S(ac) = R_S(bac) = R_S(abac) = \{aba\}$

Правые контексты

Заметим, что вместо правых контекстов, можно рассматривать множества $EndPos$ - позиций, в которых заканчивается подстрока в строке.

Пример: $S = abacaba$, $EndPos_S(aba) = \{2, 6\}$

Правые контексты

Утверждение 1. $R_S(u) = R_S(v) \Rightarrow$ одна из u, v является суффиксом другой.

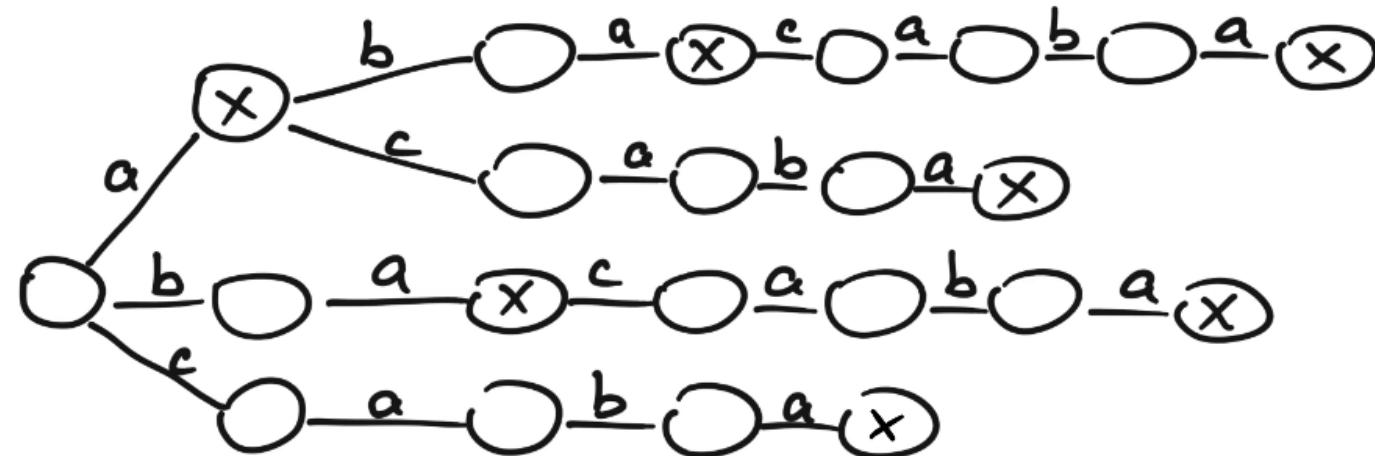
Доказательство.

Так как множества их правых контекстов совпадают, то они эти подстроки заканчиваются в одинаковых местах строки S . Выберем из u и v наименьшую по длине. Она и будет суффиксом другой. ■

Следствие. Пусть $S[a : i]$ - наибольшая по длине строка в классе эквивалентности R , а $S[b : i]$ - наименьшая. Тогда строки (и только они) $S[b : i], S[b + 1 : i], S[b + 2 : i], \dots, S[a : i]$ соответствуют R .

Правые контексты и суффиксный автомат

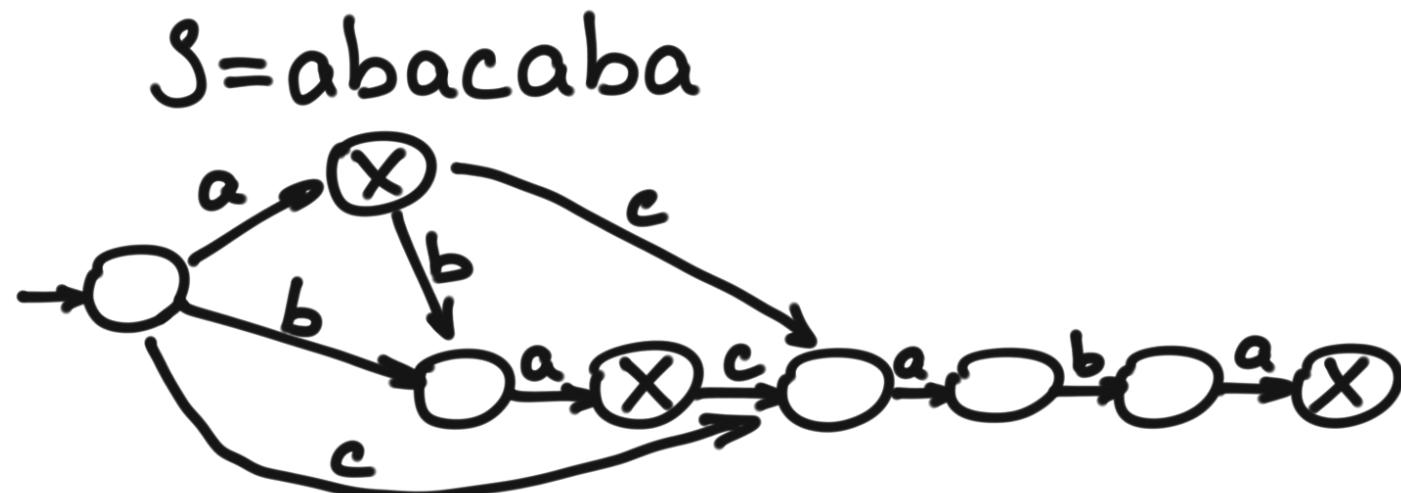
$S = abacaba$



Заметим, что правый контекст вершины задает поддерево, которое можно из нее прочитать. То есть склеиваем мы вершины с одинаковым контекстом!

Вопрос на понимание: каким контекстам соответствуют терминальные вершины?

Правые контексты и суффиксный автомат



Таким образом, число различных вершин суффиксного автомата не меньше, чем число различных контекстов (классов эквивалентности).

Хорошая новость - нижнюю границу можно достичь всегда.

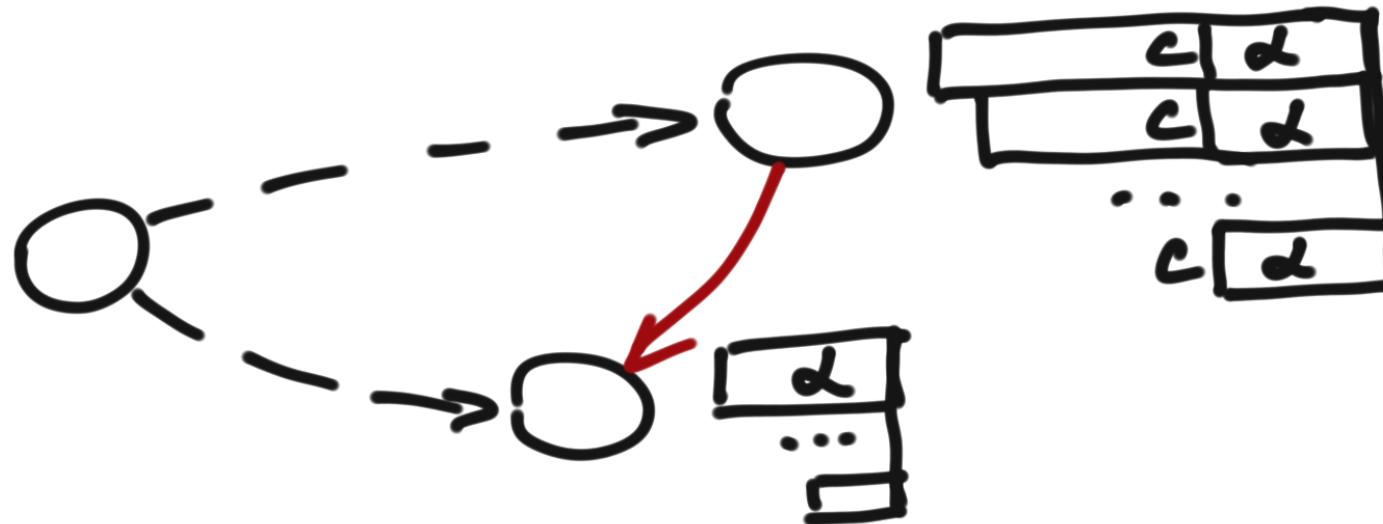
Наша цель - построить такой автомат, в котором число вершин будет равно числу классов эквивалентности.

Построение суффиксного автомата

Суффиксные ссылки

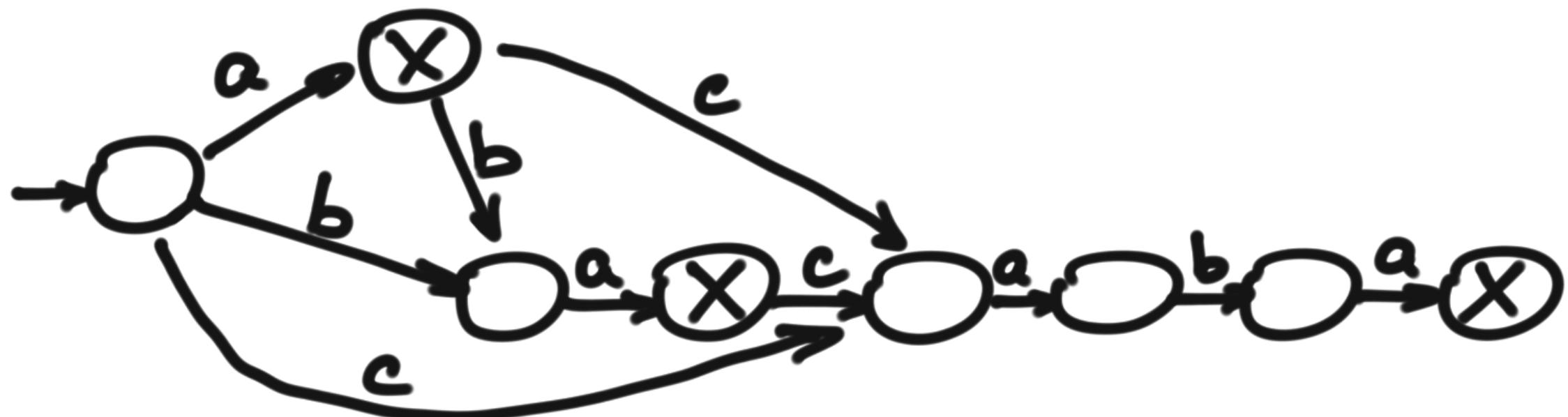
Опр. Пусть минимальная по длине строка, ведущая в вершину x , равна α .

Суффиксной ссылкой вершины x назовем ссылку, ведущую в вершину, соответствующую строке α .

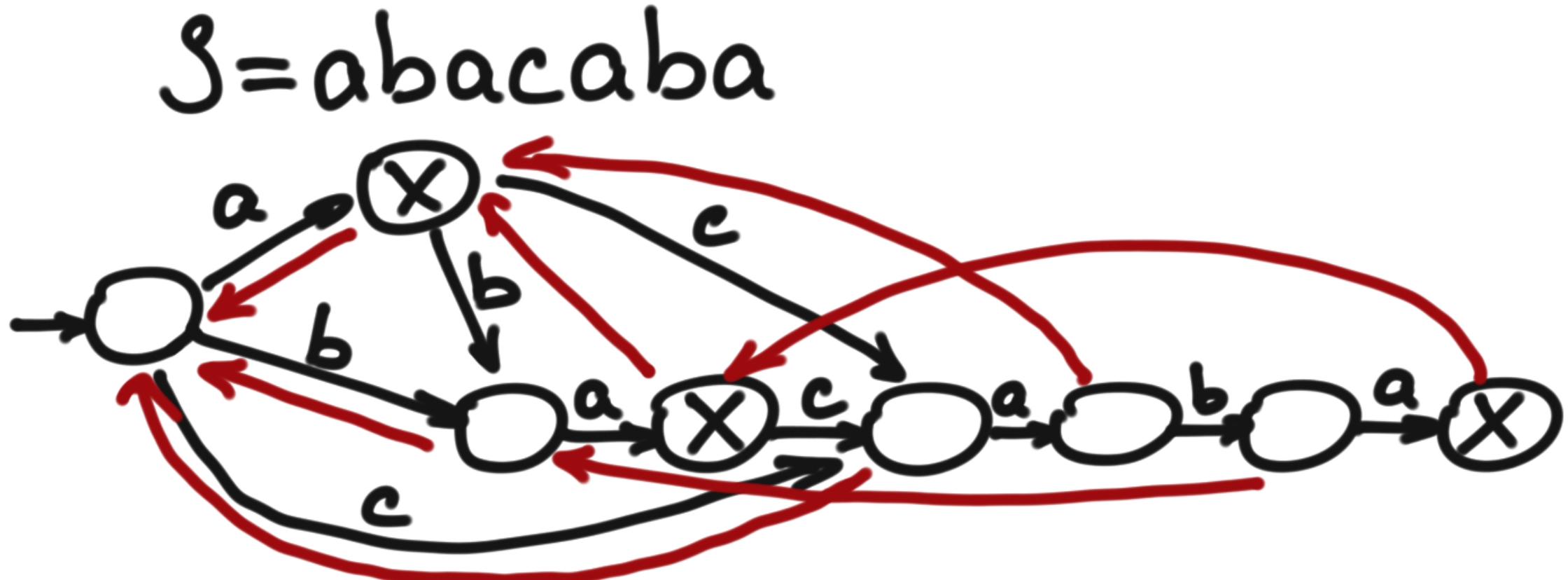


Суффиксные ссылки: пример

$s = abacaba$



Суффиксные ссылки: пример



Вершины автомата

Как и ранее, вершины будем хранить в массиве

```
Node {  
    dict[char, NodeId] transitions; // переходы  
    NodeId suffix; // суффиксная ссылка  
    size_t length; // длина наибольшей строки  
    size_t end_pos; // индекс конца данных строк  
}
```

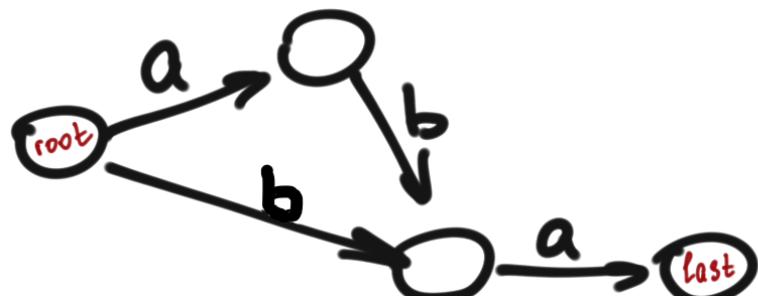
Алгоритм. Часть 1 (добрая и позитивная)

Алгоритм

Автомат будем строить итеративно, добавляя по одному символу в строку.

1. Автомат пустой строки - просто корень.
2. Пусть построен автомат для строки α . Обозначим $last$ - вершину, которая соответствует строке α (очевидно, это вершина, которая была добавлена на предыдущем шаге и единственная вершина, из которой нет переходов).
3. Осталось понять, как добавить символ c , а точнее все новые суффиксы.

$$S = aba$$



+c
+
ac
bac
abac

Новые вершины

Как изменяются множества контекстов при добавлении символа c ? Почти никак!

- Если был класс эквивалентности с контекстами $\alpha_1, \dots, \alpha_k$, то он перейдет в класс эквивалентности $\alpha_1c, \dots, \alpha_2c$ (за одним исключением, которое рассмотрим во второй части).
- А еще появится новый класс эквивалентности, соответствующий новой строке (класс эквивалентности, состоящий из пустой строки).

То есть первым делом создаем новую вершину:

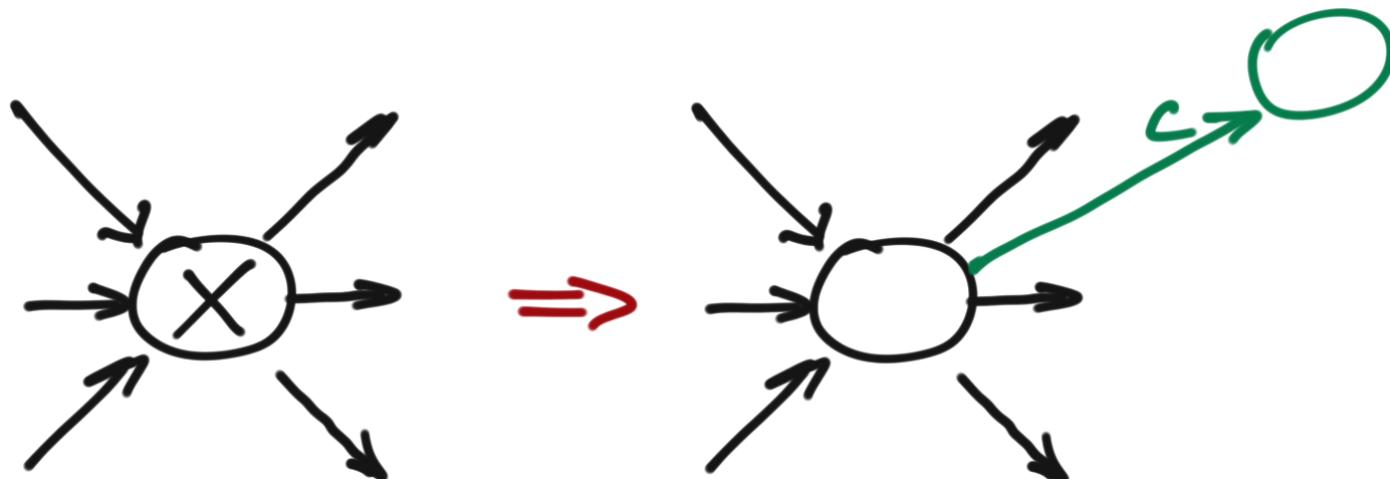
```
// добавляем i-й символ
new_node = {transitions: {}, suffix = ?, length = i + i, end_pos = i}
```

А какие переходы будут в нее вести?

Новые переходы

В новую вершину будут вести переходы из тех вершин, которые раньше содержали пустую строку в качестве контекста (почти правда, см. часть 2).

Пояснение: если у вершины был контекст $\{\alpha_1, \dots, \alpha_k, \varepsilon\}$, то теперь он станет $\{\alpha_1 c, \dots, \alpha_k c, c\}$. То есть из нее теперь можно перейти по символу c в терминальную вершину.



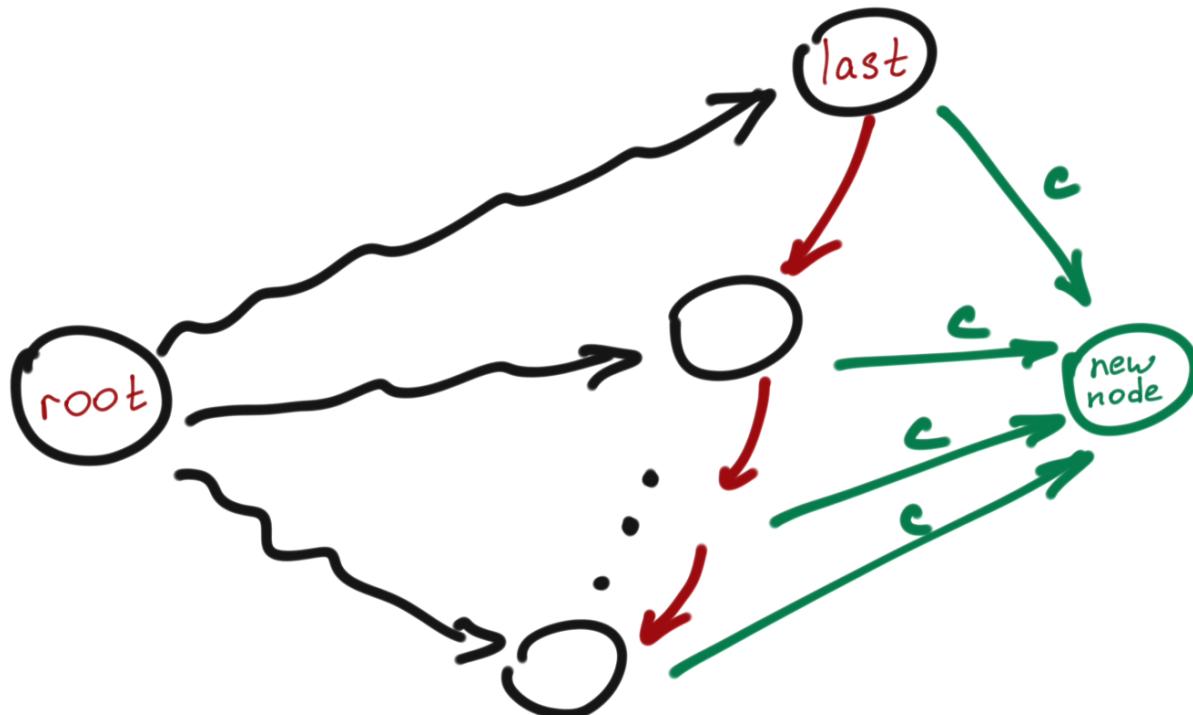
Как найти такие вершины?

Новые переходы

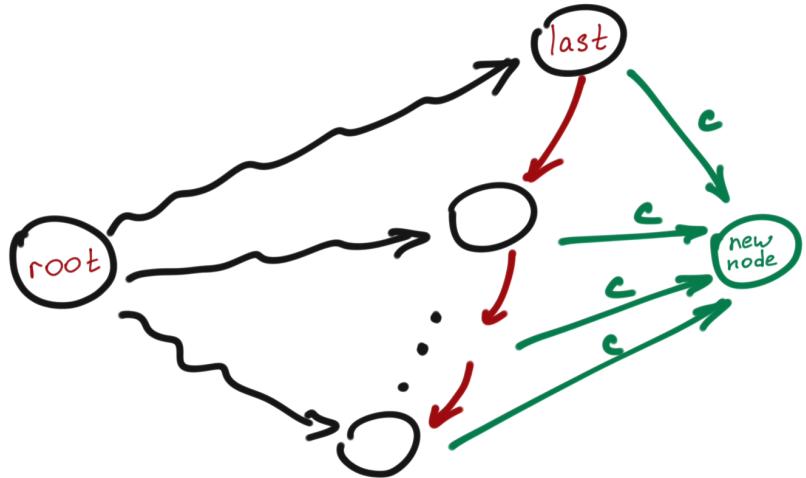
Вершина с пустой строкой в контексте = вершина, соответствующая суффиксу.

Вершина, соответствующая наибольшему суффиксу (совпадающему со всей строкой), - это вершина *last*.

А все остальные можно перебрать с помощью суффиксных ссылок!



Новые переходы



```
def AddSymbol(c):
    new_node = {{}, suffix=?, length=last.length+1, end_pos=last.end_pos+1}
    while last != None and last.transitions[c] != None:
        last.transitions[c] = new_node
        last = last.suffix
    ...
```

А что делать, если нашли суффикс, у которого уже есть переход по c ?

И куда направить суффиксную ссылку новой вершины?

Алгоритм. Часть 2 (другая)

Вопрос

Допустим, нашли суффикс α , у которого есть переход по символу c .

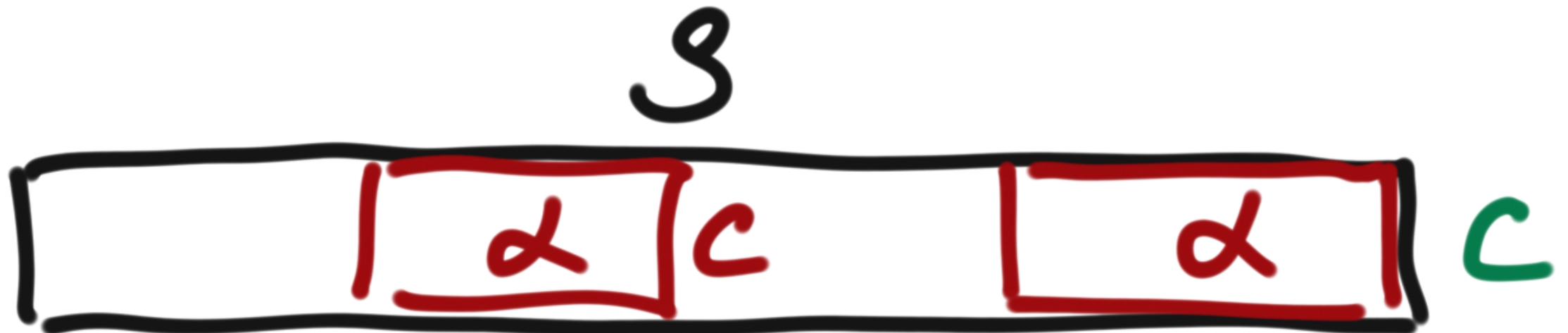
Как такое может быть?

Вопрос

Допустим, нашли суффикс α , у которого есть переход по символу c .

Как такое может быть?

Запросто. Пусть в строке S (до добавления c) был суффикс α , а также где-то внутри была подстрока αc . Значит, до состояния α можно добраться по суффиксным ссылкам от *last* и из α есть переход по c .

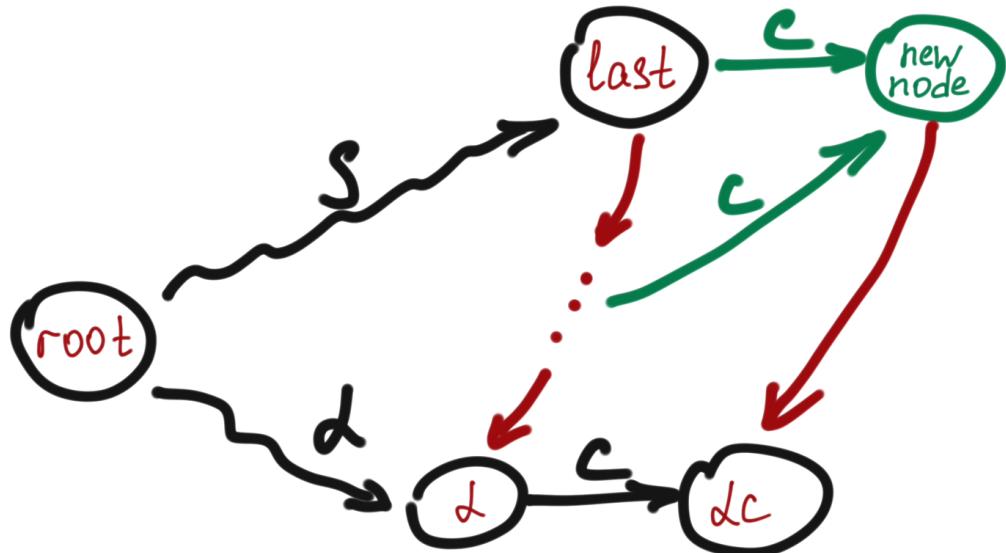


Что это значит

Допустим, нашли суффикс α , у которого есть переход по символу c .

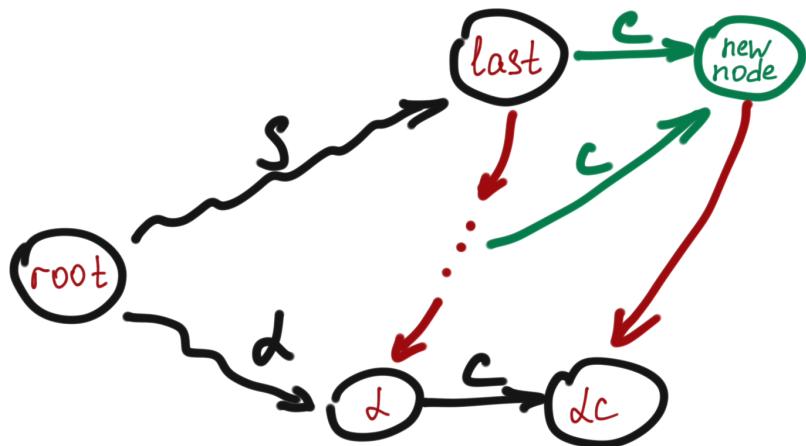
Это значит, что новый суффикс αc и все меньшие суффиксы уже лежат в суффиксном автомате!

Гипотеза: на этом можно остановиться и направить суффиксную ссылку новой вершины в вершину αc .



Гипотеза

Гипотеза: на этом можно остановиться и направить суффиксную ссылку новой вершины в вершину αc .



```
def AddSymbol(c):
    new_node = {}, suffix=?, length=last.length+1, end_pos=last.end_pos+1}
    while last != None and last.transitions[c] != None:
        last.transitions[c] = new_node
        last = last.suffix
    new_node.suffix = (last == None ? root : last)
    last = new_node
```

Гипотеза: на этом можно остановиться и направить суффиксную ссылку новой вершины в вершину αc .

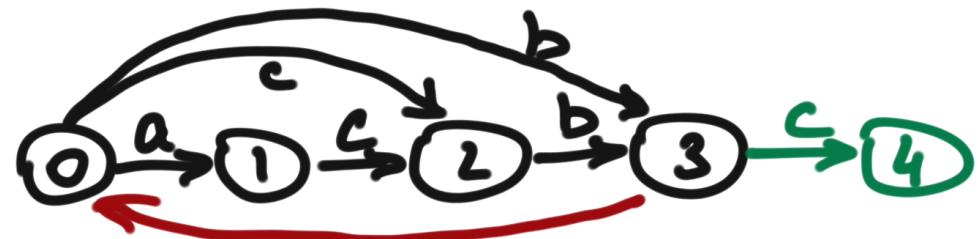
Проблема в том, что этой вершине может соответствовать не только строка αc , но и другие подстроки, которые не являются суффиксами sc .

Пример:

Вершина соответствующая новому суффиксу c (2) уже есть в автомате. Но этой вершине также соответствует строка ac , которая не является суффиксом.

Иными словами, у строки c появился новый контекст ε , а у строки ac - нет.

$$S = acb + c$$



Вывод

Если появился новый суффикс αs , который раньше был просто подстрокой s , то это может привести к появлению нового класса эквивалентности.

А точнее к "расщеплению" класса эквивалентности строки αs на

- строки, которые являются суффиксами (у которых в контексте появляется ε)
- и остальные (контекст которых не меняется)

Случай 1 (простой)

Опр. Назовем переход из $node_1$ в $node_2$ в суффавтомате *сплошным (solid)*, если $node_1.length + 1 == node_2.length$

Смысл: наибольшая строка в $node_2$ получается дописыванием одного символа к наибольшей строке в $node_1$.

Зачем это надо: если по суффиксным ссылкам пришли от $last$ пришли к α , из которой есть сплошной переход в αc , то расщеплять ничего не нужно!

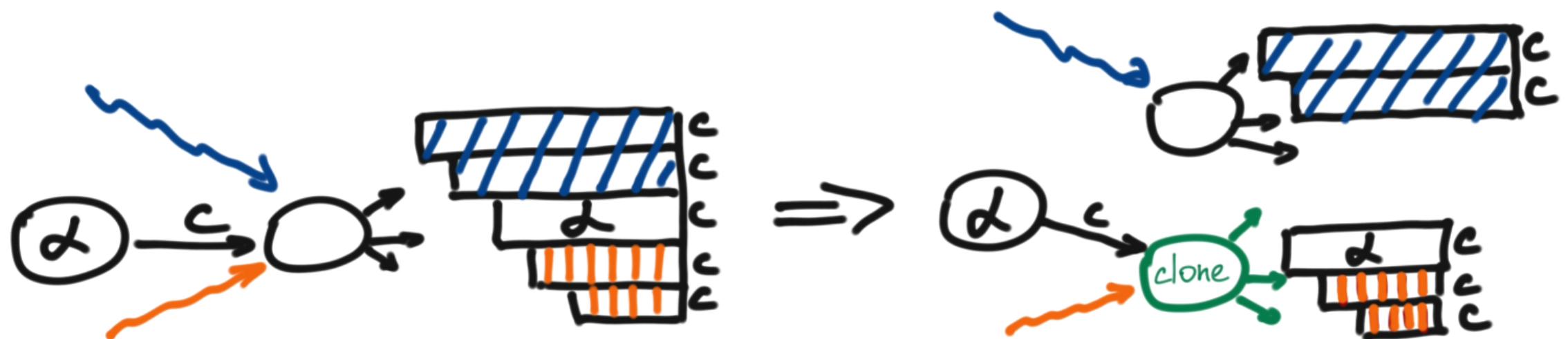
```
def AddSymbol(c):
    new_node = {}, suffix=?, length=last.length+1, end_pos=last.end_pos+1}
    while last != None and last.transitions[c] != None:
        last.transitions[c] = new_node
        last = last.suffix
    if last == None:
        new_node.suffix = root
    else if last.length + 1 == last.transitions[c].length: # переход сплошной
        new_node.suffix = last.transitions[c]
    else:
        ...
    last = new_node
```

Случай 2 (клонирование)

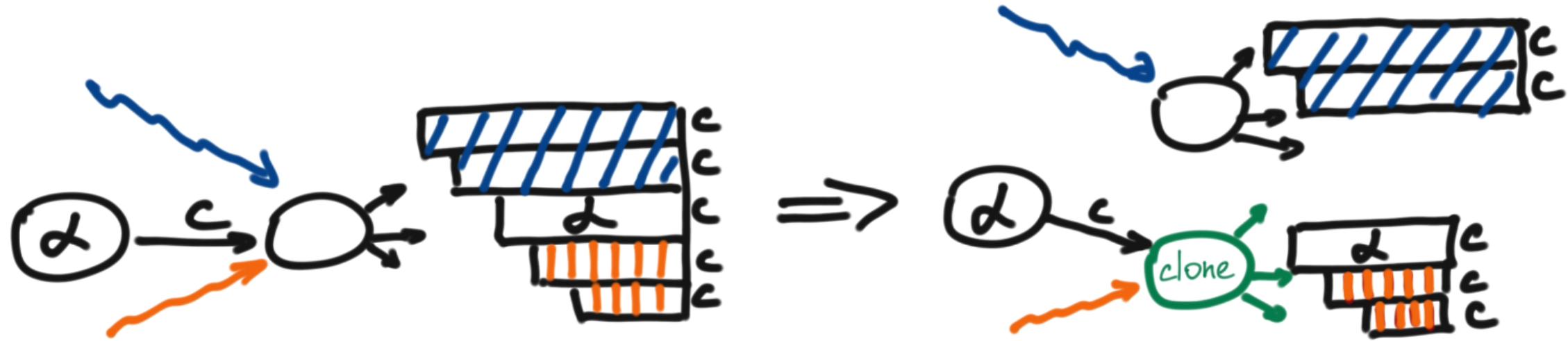
Если переход не сплошной, то состояние αc придется расщепить.

Помним, что состоянию соответствует некоторая наибольшая строка + некоторая непрерывная последовательность ее наибольших суффиксов.

Таким образом, все строки длиннее αc останутся в старом состоянии, а остальные пойдут в новое (все они - суффиксы новой строки!).



Случай 2 (клонирование)



Что значит "расщепить"/"клонировать"?

- Ясно, что все переходы исходной вершины останутся и у новой. То есть переходы просто копируем.
- Суффиксная ссылка клона будет такой же, как и у исходной.
- Суффиксная ссылка исходной вершины теперь ведет в клона.
- Переходы из α и ее суффиксов теперь должны вести в клона.

Случай 2 (клонирование)

- Ясно, что все переходы исходной вершины останутся и у новой. То есть переходы просто копируем.
- Суффиксная ссылка клона будет такой же, как и у исходной.
- Суффиксная ссылка исходной вершины теперь ведет в клона.
- Переходы из α и ее суффиксов теперь должны вести в клона.

```
def Clone(parent, c, node):
    clone = {node.transitions, node.suffix, length=parent.length+1}
    node.suffix = clone
    while parent != None and parent.transitions[c] == node:
        parent.transitions[c] = clone
        parent = parent.suffix
    return clone
```

Итог

$last$ - вершина, соответствующая текущей строке S

- При добавлении символа c создаем вершину (соответствует новой строке).
- Идем по суффиссылкам $last$ и создаем переходы по c в новую вершину.
(обновляем старые суффиксы, дописывая к ним новый символ)
- Если встретили суффикс, который уже содержит переход по c , это значит этот суффикс уже есть в автомате.
 - 1 случай. Если переход сплошной, направляем суффиссылку новой вершины в вершину, в которую ведет переход.
 - 2 случай. Иначе запускаем процедуру *Clone*. Направляем суффиссылку новой вершины в клона.
- Иначе направляем суффиксную ссылку новой вершины в корень.
(все суффиксы обновленной строки ведут в новую вершину)
- Обновляем $last = new_node$.

Итог

```
def Clone(parent, c, node):
    clone = {node.transitions, node.suffix, length=parent.length+1}
    node.suffix = clone
    while parent != None and parent.transitions[c] == node:
        parent.transitions[c] = clone
        parent = parent.suffix
    return clone
```

```
def AddSymbol(c):
    new_node = {}, suffix=?, length=last.length+1, end_pos=last.end_pos+1}
    while last != None and last.transitions[c] != None:
        last.transitions[c] = new_node
        last = last.suffix
    if last == None:
        new_node.suffix = root
    else if last.length + 1 == last.transitions[c].length: # переход сплошной
        new_node.suffix = last.transitions[c]
    else:
        new_node.suffix = Clone(last, c, last.transitions[c])
    last = new_node
```

Анализ памяти

- Добавление символа приносит не более $2x$ вершин (`new_node` , `clone`).
Таким образом, число вершин $\Theta(|S|)$
- В каждую вершину ведет ровно 1 сплошной переход (если наибольшая строка у вершины это αc , то в нее есть переход по c из вершины со строкой α). Значит таких переходов $\Theta(|S|)$.
- Каждому несплошному переходу поставим в соответствие длиннейший путь, которому он принадлежит. Самый длинный путь обязательно закончится в терминальной вершине (иначе он не самый длинный, так как можно пройти дальше). Следовательно, этот путь соответствует суффиксу строки S . Но каждому суффиксу соответствует ровно 1 терминальный путь в автомате. Следовательно, число несплошных переходов $\Theta(|S|)$

Таким образом, суммарные затраты памяти $\Theta(|S|)$.

Анализ времени

Проанализируем суммарное время работы циклов при добавлении символов (циклы `while` в `AddSymbol` и `Clone`).

Ситуация аналогична алгоритмам построения префикс-функции и Ахо-Корасик.

На каждой итерации `while` глубина (`last` и `parent` соответственно) уменьшается на 1. Но добавление символа увеличивает глубину ровно на 1 за вызов функции `AddSymbol`.

Ясно, что число уменьшений глубины n_- не больше числа ее увеличений n_+ . То есть $n_- \leq n_+ \leq |S|$.