Экспресс-анализ потоковых текстовых данных на предмет вхождения в них ключевых слов и фраз.
Часть I: исследование алгоритмов поиска ключевых слов.

В.А. Васенин (vasenin@msu.ru), НИИ Механики МГУ
М.Д. Дзабраев (dzabraew@gmail.com), Мех-Мат МГУ
В.А. Роганов (var@msu.ru), НИИ Механики МГУ

27 октября 2015

План доклада

1. Постановка решаемой задачи.
2. Обзор методов решения.
3. Классические подходы.
4. Особенности нашего подхода.
5. Сравнение производительности.
6. Выводы и перспективы.

Базовые определения

Σ — алфавит

Определение. Конечную последовательность букв будем назвать словом.

Определение. Будем говорить, что слово P_i входит в T, если $T = u + P_i + v$, где операция $+$ есть конкатенация строк, а u, v — слова из алфавита Σ, возможно пустые.

Определение. Будем называть слово T текстом.

Определение. Слова P_i будем называть шаблонами.

Базовые определения

Определение. Скоростью программной реализации алгоритма A на тексте T будем называть величину $H_A(T) = \frac{\text{size}(T)}{t}$, где $\text{size}(T)$ есть размер текста T в битах, а t — время в секундах, затраченное на поиск шаблонов $\{P_i\}$ в тексте T.

Постановка решаемой задачи. Требуется с максимальной скоростью на доступном оборудовании производить поиск ключевых слов и фраз (шаблонов).

Измерять скорость будем на нескольких тестовых файлах.

Обзор

Обзор

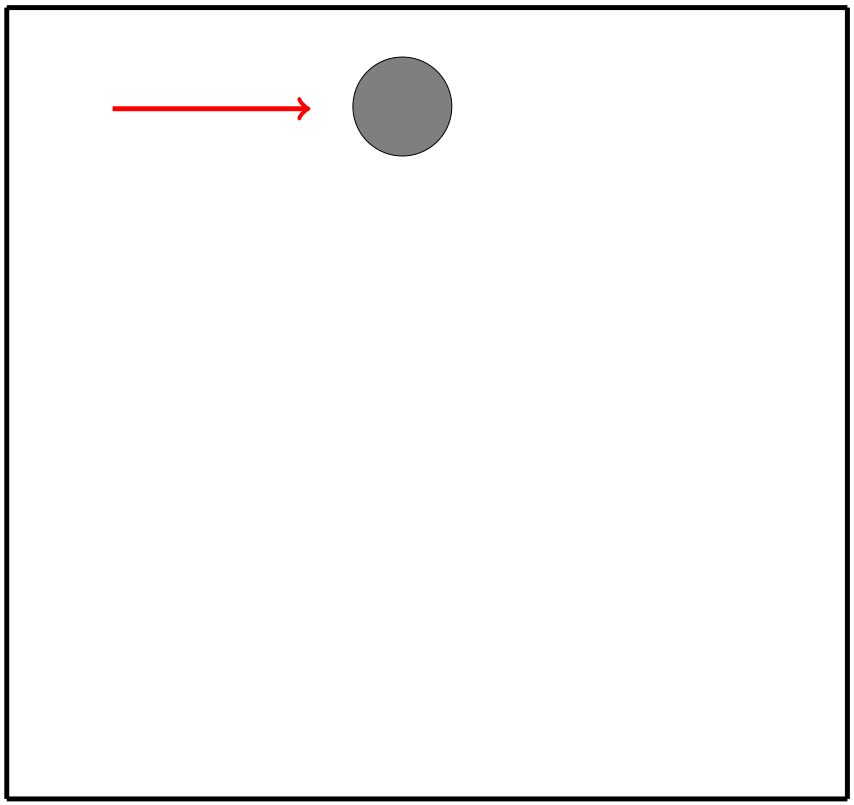
На сегодняшний день известны подходы для поиска множественных шаблонов, которые используют:

- DFA
- NFA
- Алгоритмы, использующие хэширование.

Существуют реализации программ для поиска шаблонов на разной аппаратуре:

- CUDA
- ASIC application-specific integrated circuit
- FPGA field-programmable gate array

Алгоритм Aho-Corasick, построение дерева



Шаблоны:

bab

bc

bca

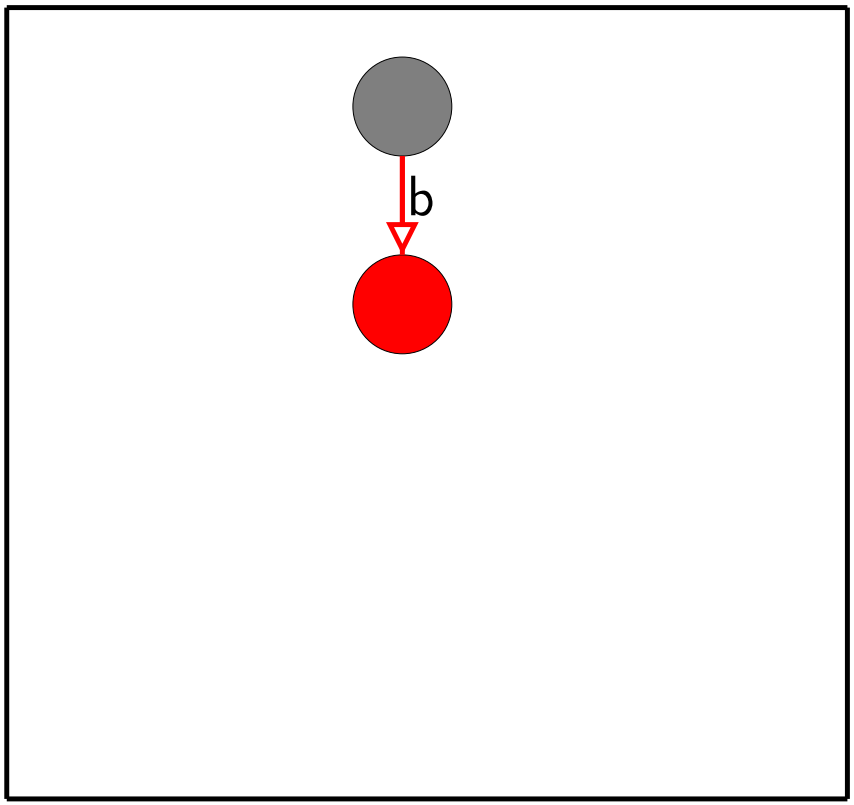
a

ab

a

acc

Алгоритм Aho-Corasick, построение дерева



Шаблоны:

→ bab

bc

bca

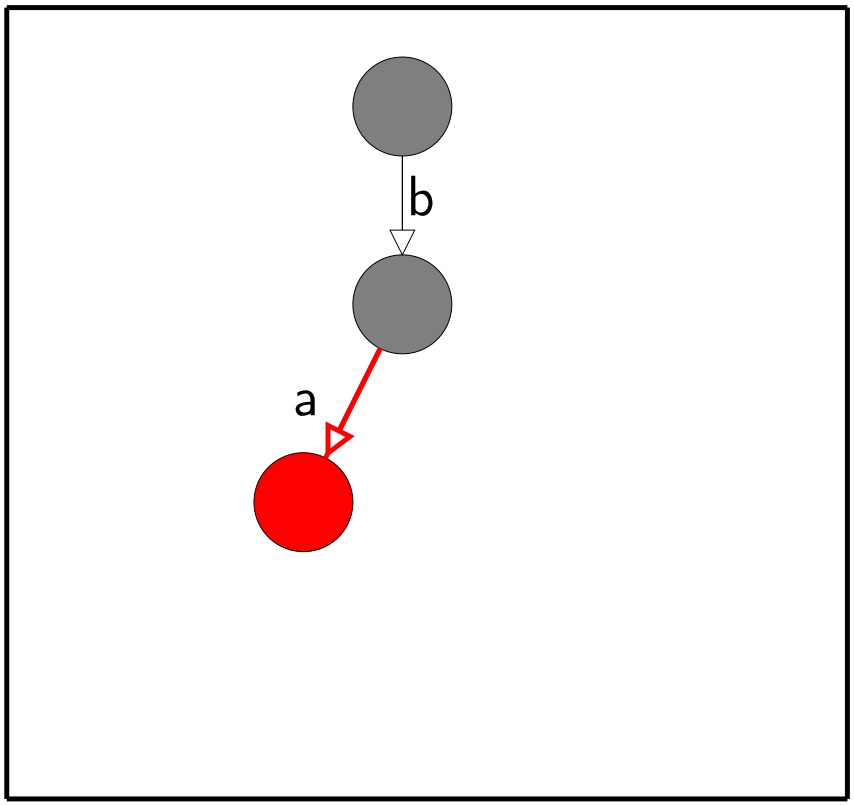
a

ab

a

caa

Алгоритм Aho-Corasick, построение дерева



Шаблоны:

→ bab

bc

bca

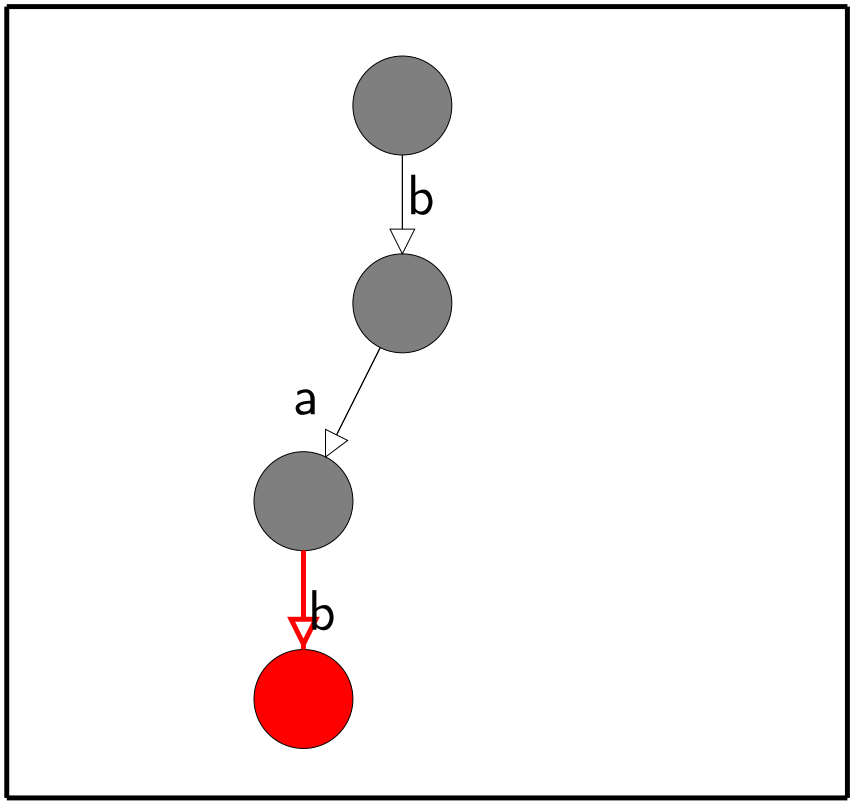
a

ab

a

caa

Алгоритм Aho-Corasick, построение дерева



Шаблоны:

→ bab

bc

bca

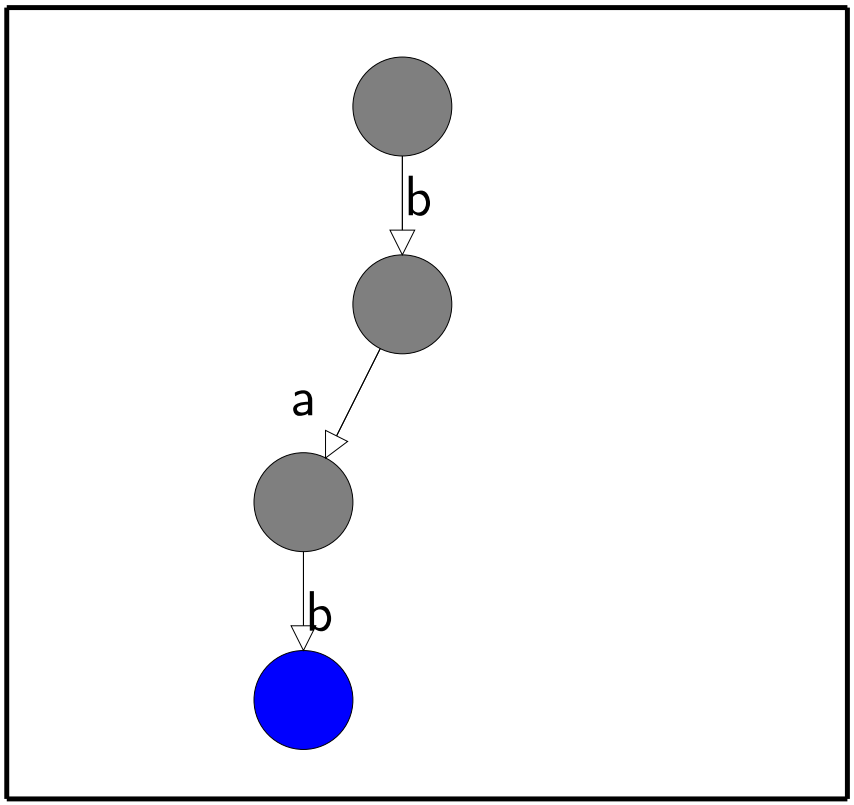
a

ab

a

caa

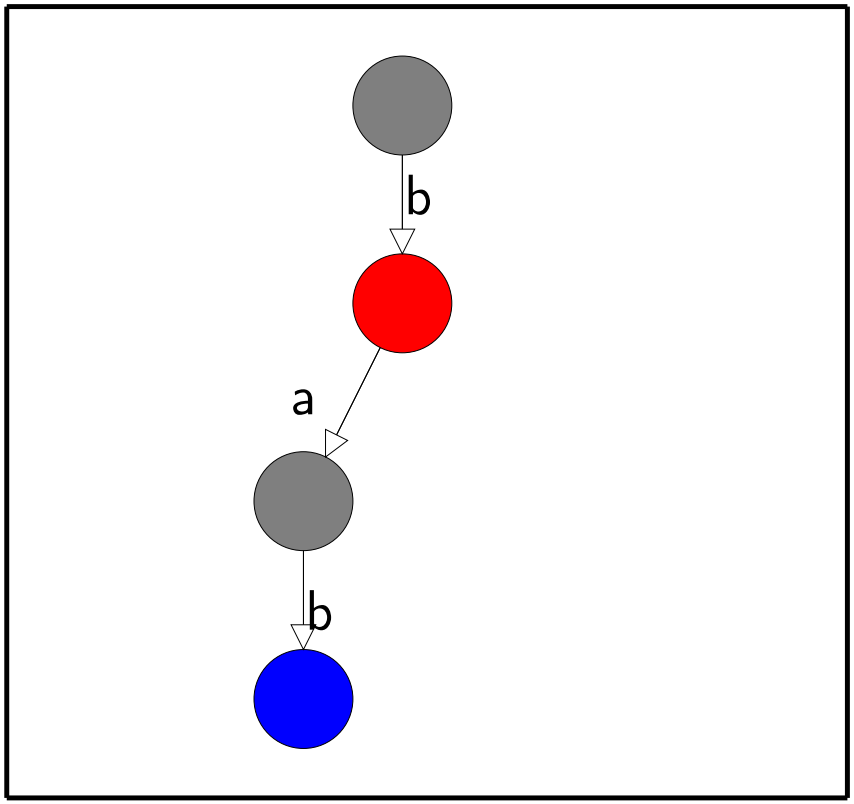
Алгоритм Aho-Corasick, построение дерева



Шаблоны:

- → bab
- bc
- bca
- a
- ab
- a
- caa

Алгоритм Aho-Corasick, построение дерева



Шаблоны:

bab

→ bc

bca

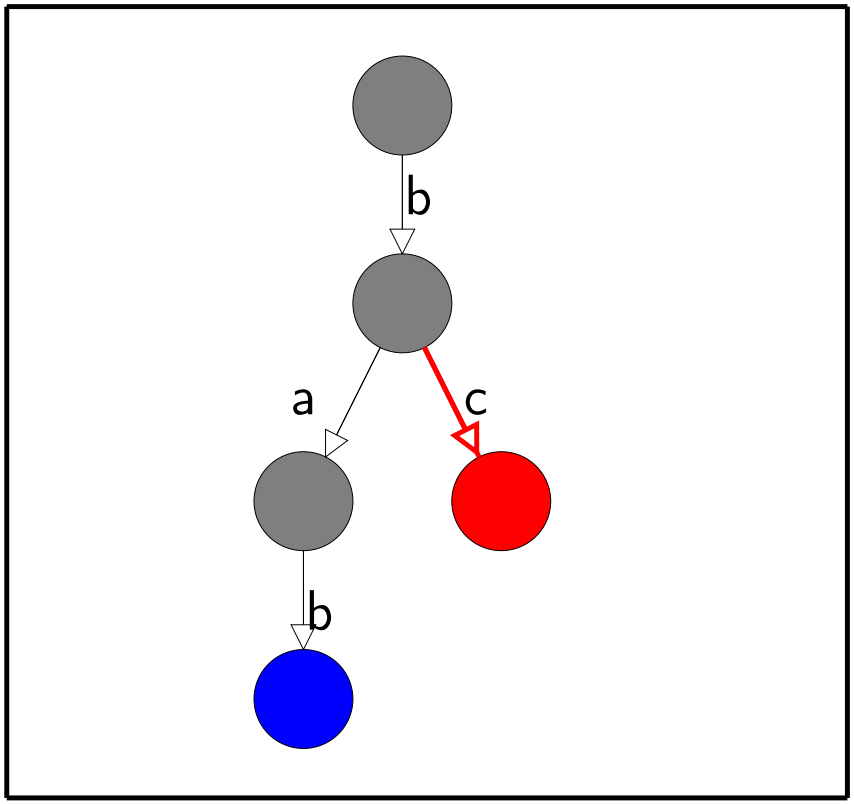
a

ab

a

caa

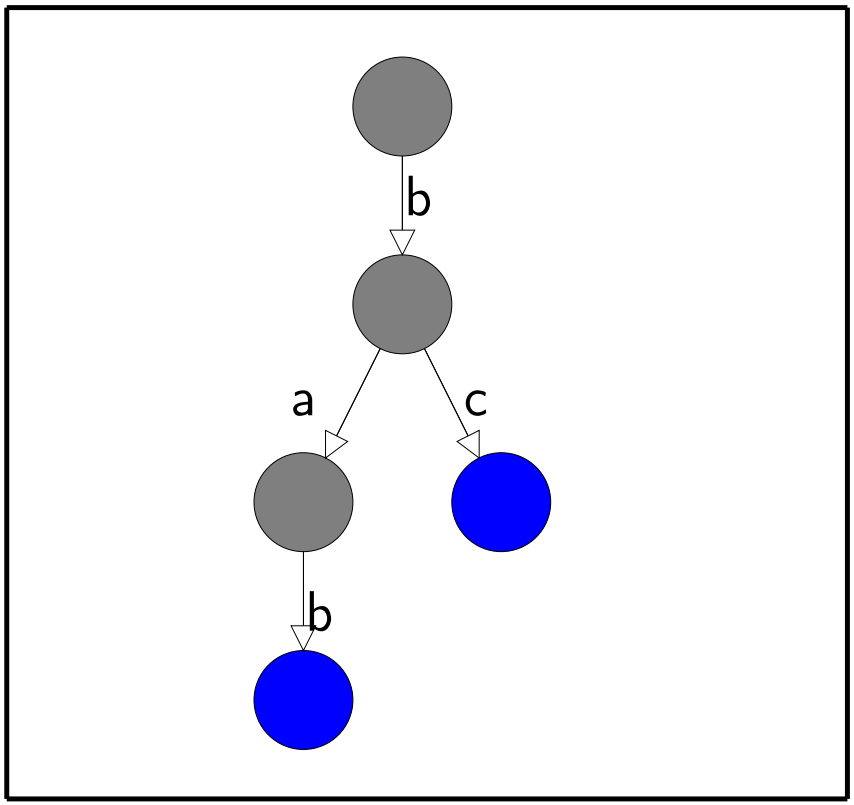
Алгоритм Aho-Corasick, построение дерева



Шаблоны:

- bab
- → bc
- bca
- a
- ab
- a
- caa

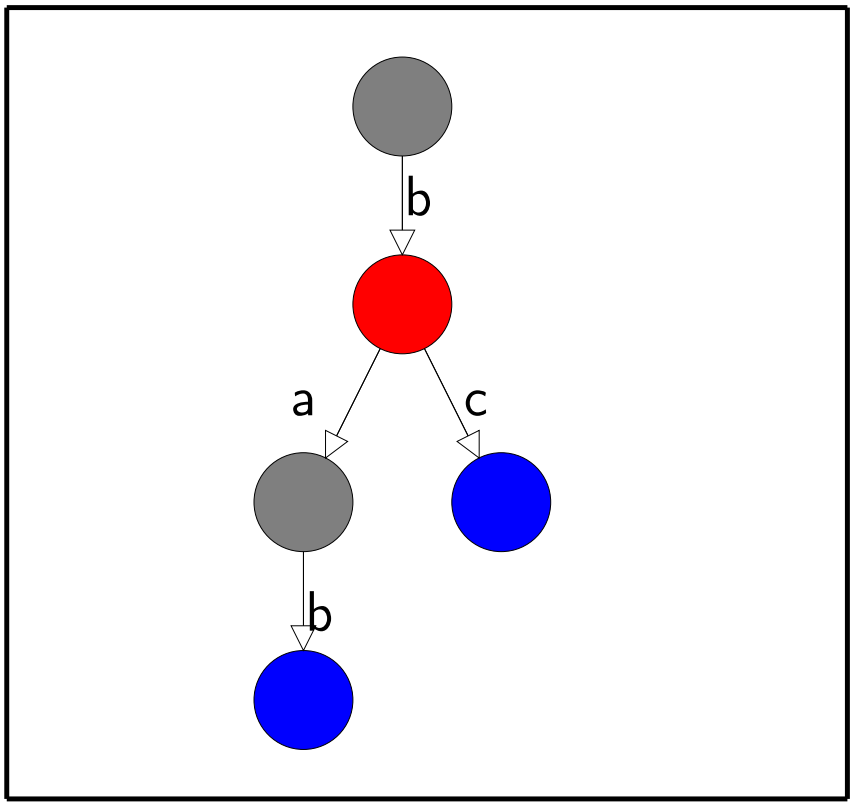
Алгоритм Aho-Corasick, построение дерева



Шаблоны:

- bab
- → bc
- bca
- a
- ab
- a
- caa

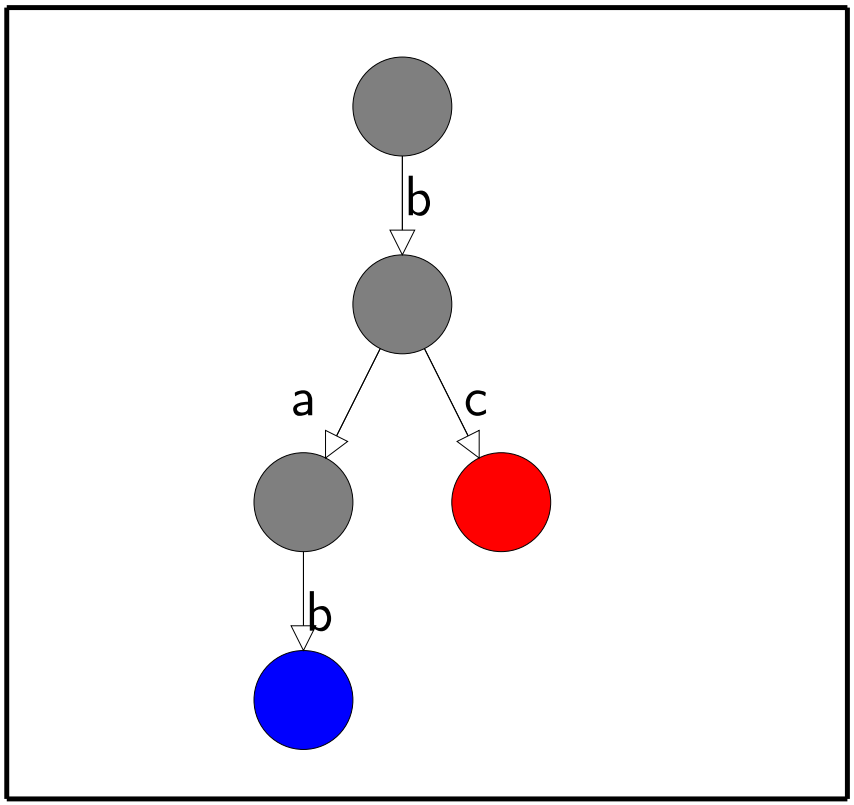
Алгоритм Aho-Corasick, построение дерева



Шаблоны:

- bab
- bc
- → bca
- a
- ab
- a
- caa

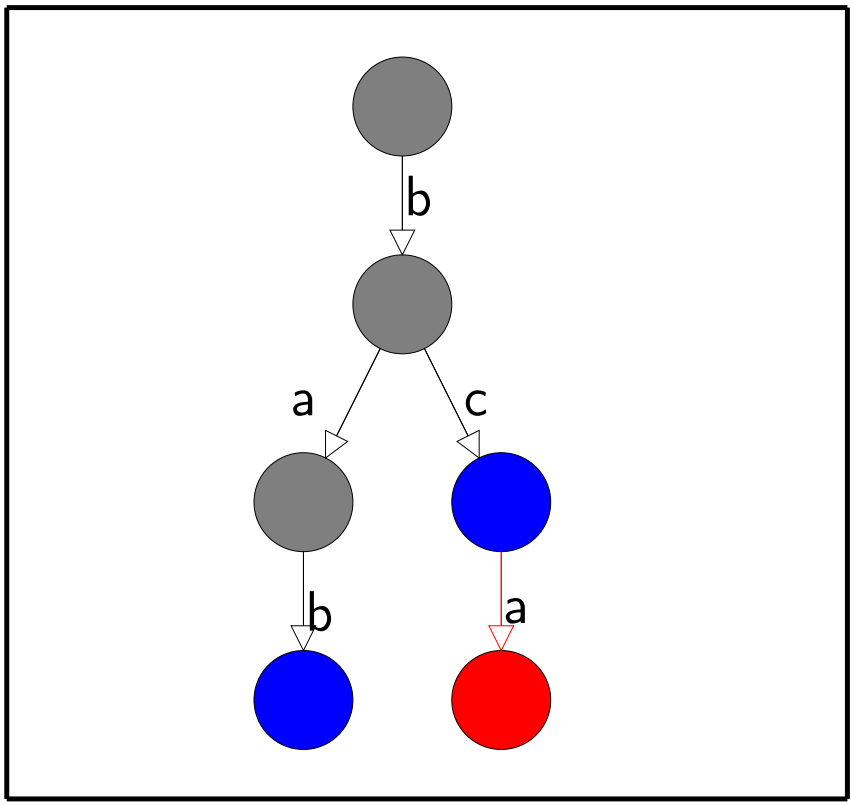
Алгоритм Aho-Corasick, построение дерева



Шаблоны:

- bab
- bc
- → bca
- a
- ab
- a
- caa

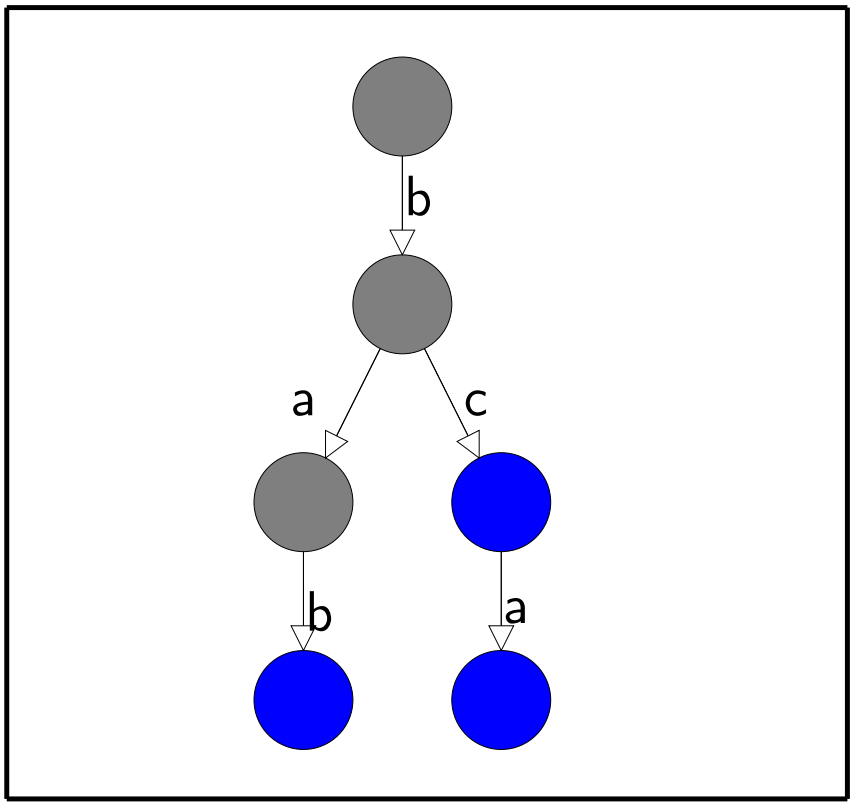
Алгоритм Aho-Corasick, построение дерева



Шаблоны:

- bab
- bc
- → bca
- a
- ab
- a
- caa

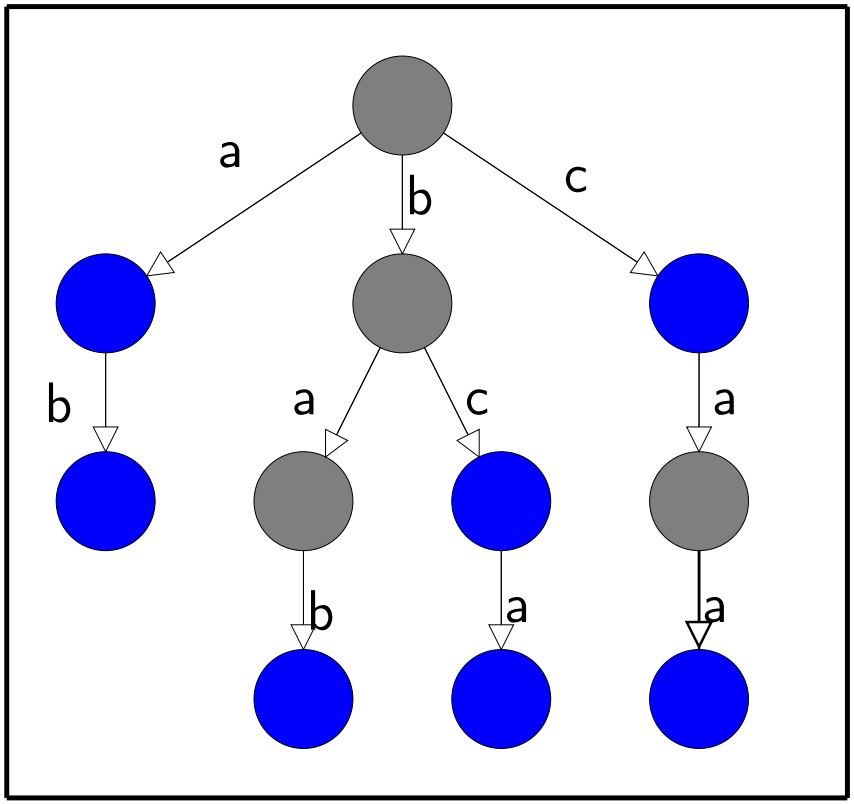
Алгоритм Aho-Corasick, построение дерева



Шаблоны:

- bab
- bc
- → bca
- a
- ab
- a
- caa

Алгоритм Aho-Corasick, построение дерева



Шаблоны:

bab

bc

bca

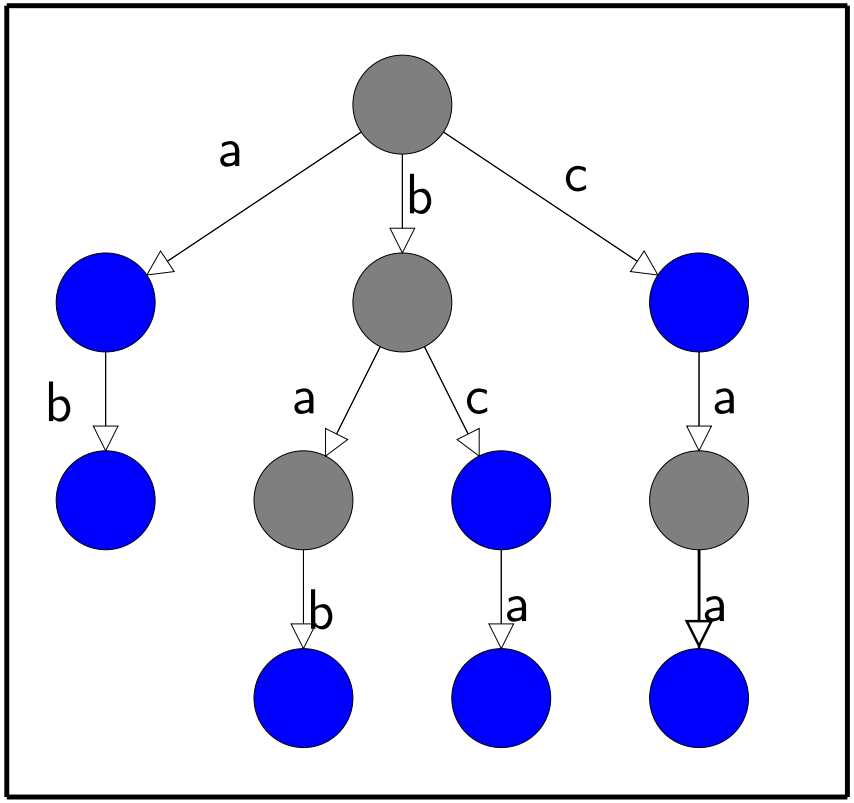
a

ab

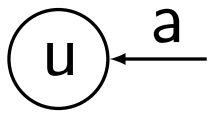
a

→ caa

Алгоритм Aho-Corasick, создание суффиксных ссылок



Определение. l(v)=



l(u)=a

l(r)=ε

Определение. w(v)=

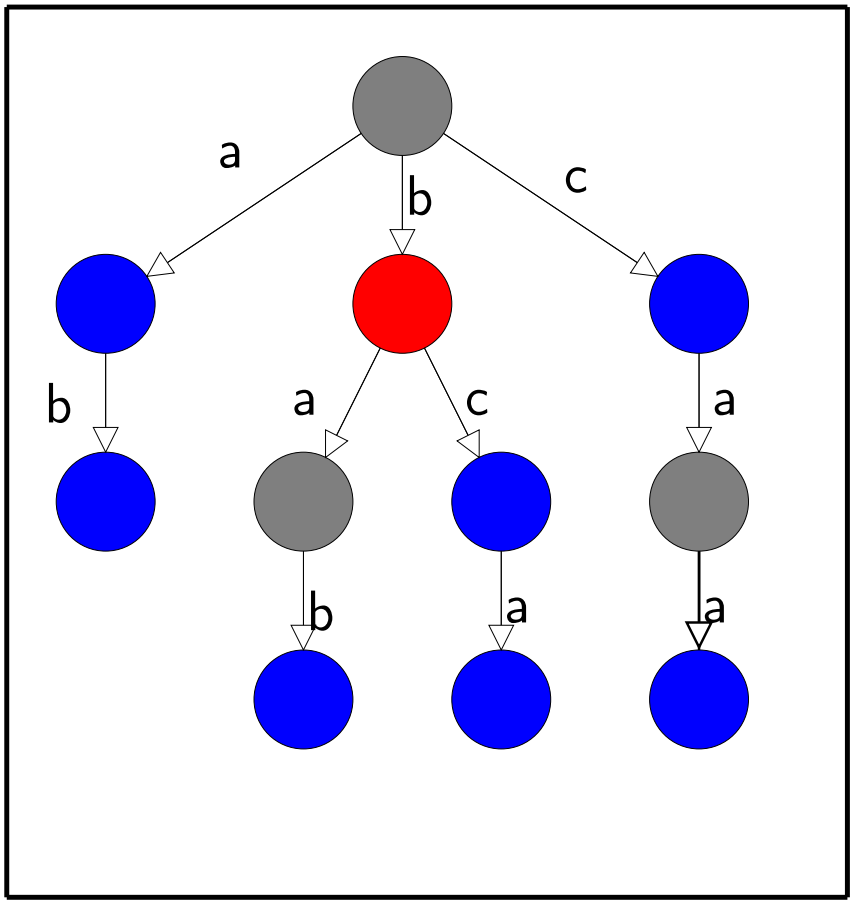
$r = v_0 \to v_1 \to \ldots \to v_n = v$

w(v)=l(v_1)..l(v_n)

Определение.

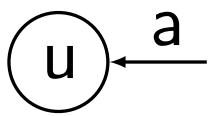
Суффиксная ссылка для u — это (u,v), где w(u)=s+w(v), и w(v) самый длинный суффикс w(u). $u \neq v$.

Алгоритм Aho-Corasick, создание суффиксных ссылок



$b = b + \varepsilon$

Определение. l(v)=



l(u)=a

$l(r)=\varepsilon$

Определение. w(v)=

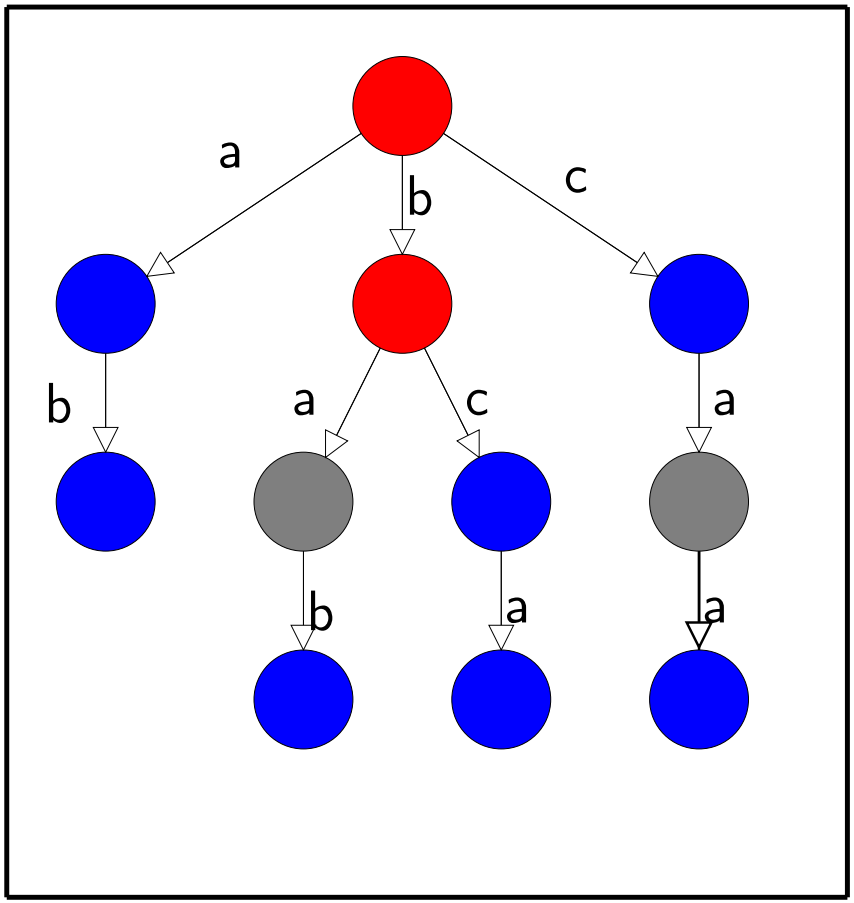
$r = v_0 \to v_1 \to \ldots \to v_n = v$

$w(v)=l(v_1)..l(v_n)$

Определение.

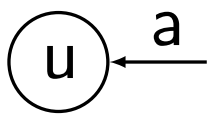
Суффиксная ссылка для u — это (u,v), где $w(u)=s+w(v)$, и $w(v)$ самый длинный суффикс $w(u)$. $u \neq v$.

Алгоритм Aho-Corasick, создание суффиксных ссылок



$b = b + \varepsilon$

Определение. l(v)=



l(u)=a

$l(r)=\varepsilon$

Определение. w(v)=

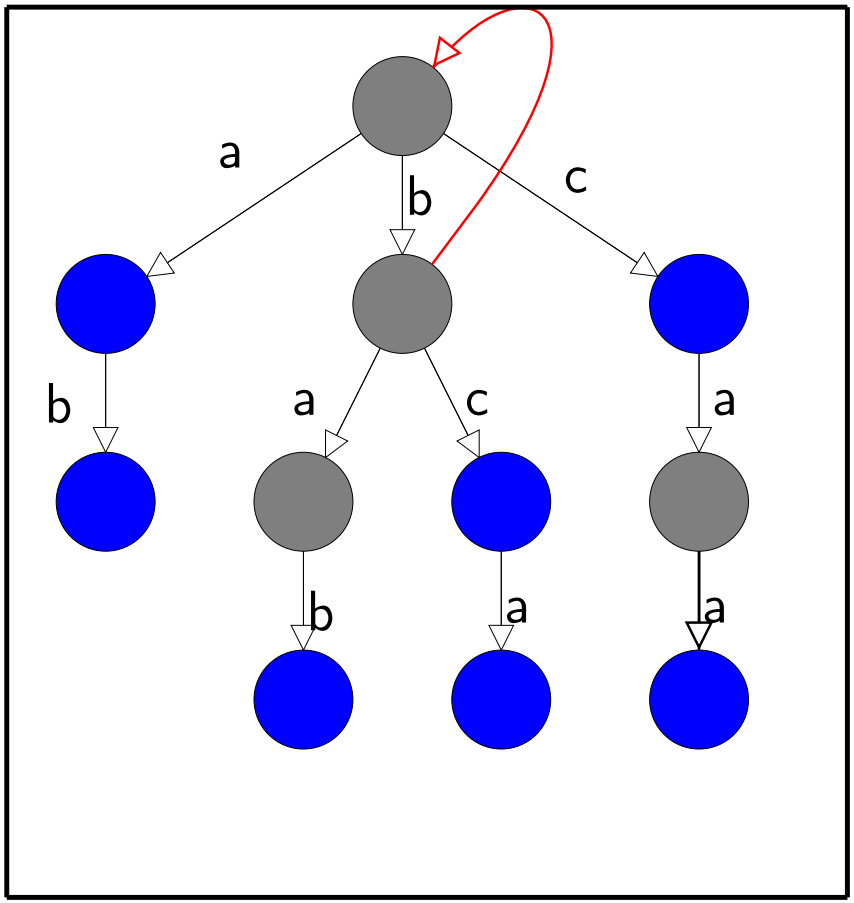
$r = v_0 \to v_1 \to \ldots \to v_n = v$

$w(v)=l(v_1)..l(v_n)$

Определение.

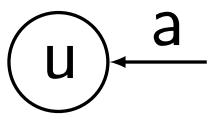
Суффиксная ссылка для u — это (u,v), где $w(u)=s+w(v)$, и $w(v)$ самый длинный суффикс $w(u)$. $u \neq v$.

Алгоритм Aho-Corasick, создание суффиксных ссылок



b = b + ε

Определение. l(v)=



l(u)=a

l(r)=ε

Определение. w(v)=

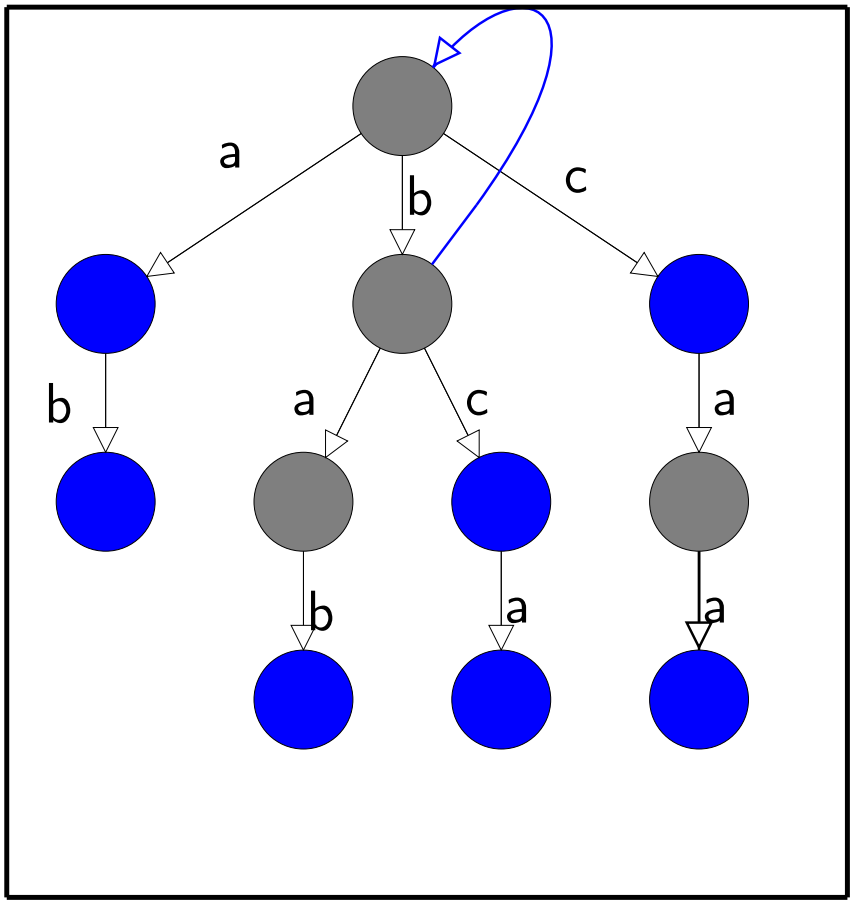
$r = v_0 \to v_1 \to \ldots \to v_n = v$

w(v)=l(v_1)..l(v_n)

Определение.

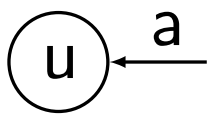
Суффиксная ссылка для u — это (u,v), где w(u)=s+w(v), и w(v) самый длинный суффикс w(u). $u \neq v$.

Алгоритм Aho-Corasick, создание суффиксных ссылок



$b = b + \varepsilon$

Определение. l(v)=



l(u)=a

$l(r)=\varepsilon$

Определение. w(v)=

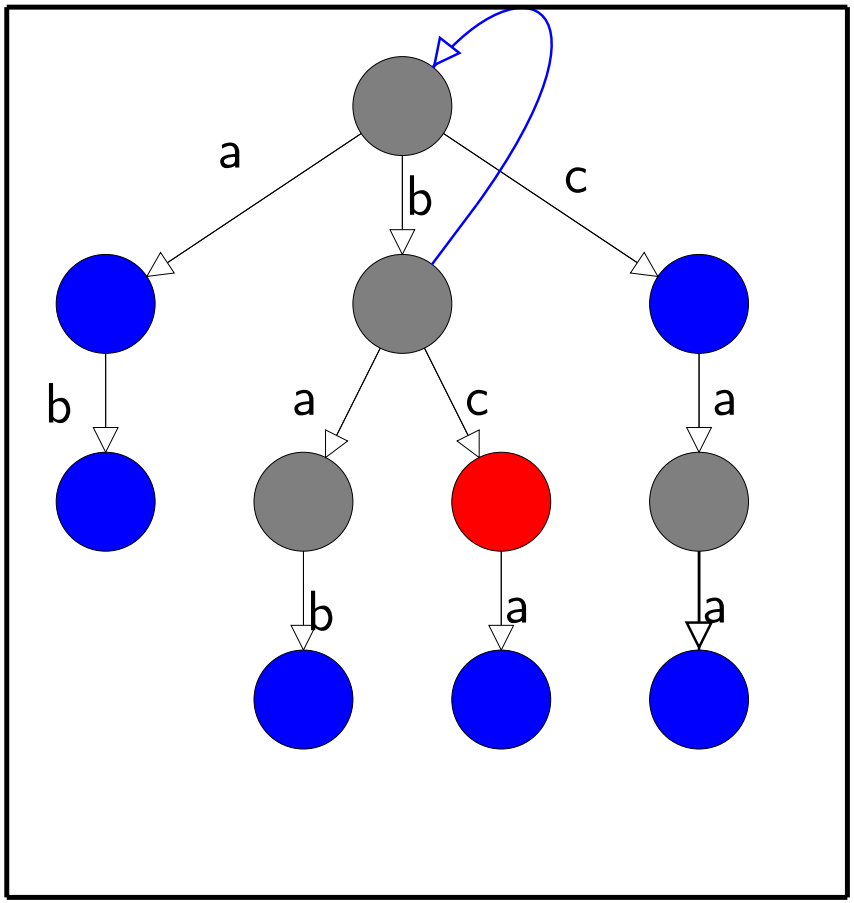
$r = v_0 \to v_1 \to \ldots \to v_n = v$

$w(v)=l(v_1)..l(v_n)$

Определение.

Суффиксная ссылка для u — это (u,v), где $w(u)=s+w(v)$, и $w(v)$ самый длинный суффикс $w(u)$. $u \neq v$.

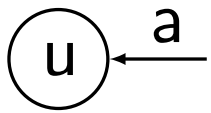
Алгоритм Aho-Corasick, создание суффиксных ссылок



$b = b + \varepsilon$

$bc = b + c$

Определение. l(v)=



l(u)=a

l(r)=ε

Определение. w(v)=

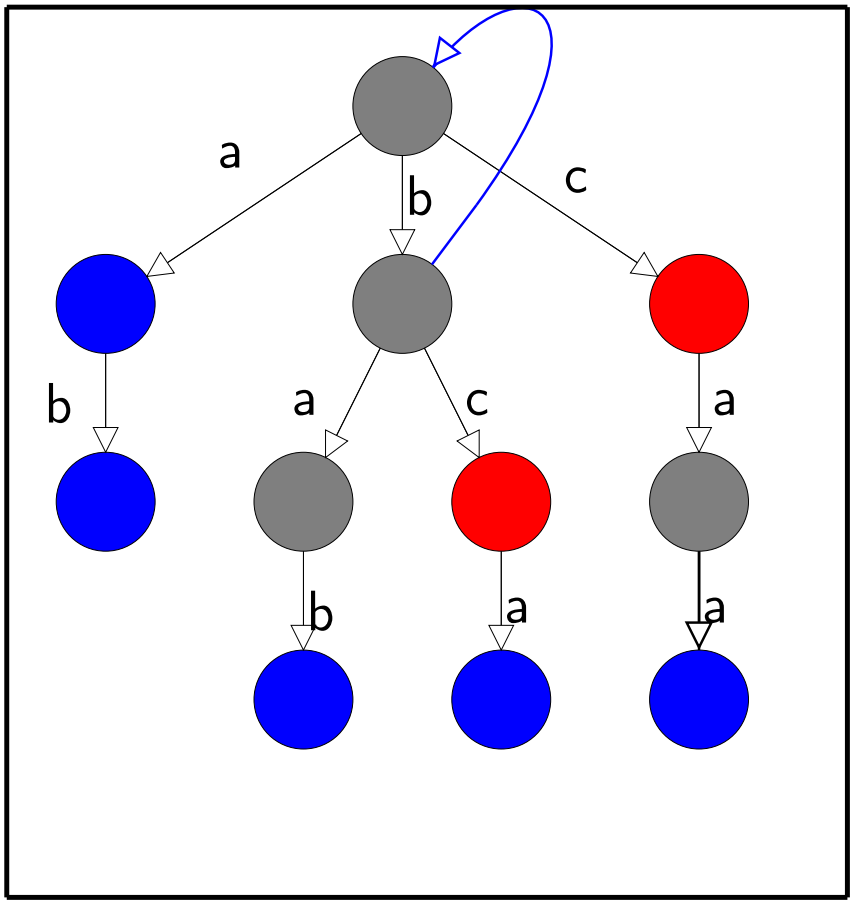
$r = v_0 \to v_1 \to \ldots \to v_n = v$

$w(v)=l(v_1)..l(v_n)$

Определение.

Суффиксная ссылка для u — это (u,v), где $w(u)=s+w(v)$, и $w(v)$ самый длинный суффикс $w(u)$. $u \neq v$.

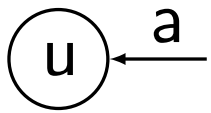
Алгоритм Aho-Corasick, создание суффиксных ссылок



$b = b + \varepsilon$

$bc = b + c$

Определение. l(v)=



l(u)=a

$l(r)=\varepsilon$

Определение. w(v)=

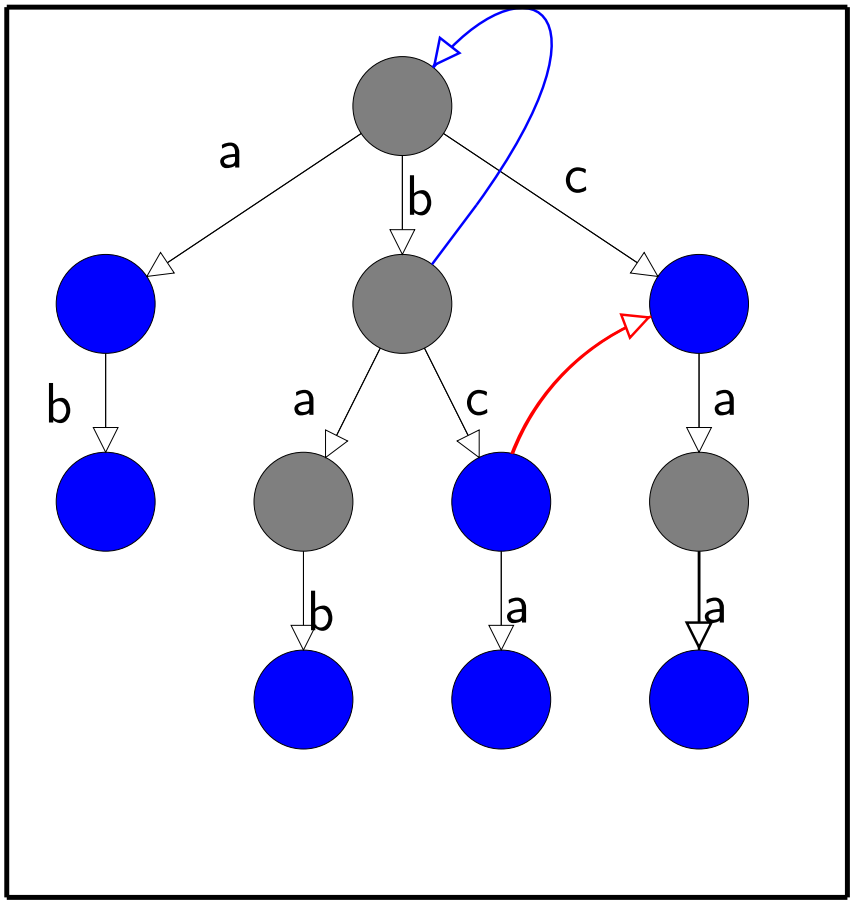
$r = v_0 \to v_1 \to \ldots \to v_n = v$

$w(v)=l(v_1)..l(v_n)$

Определение.

Суффиксная ссылка для u — это (u,v), где $w(u)=s+w(v)$, и $w(v)$ самый длинный суффикс $w(u)$. $u \neq v$.

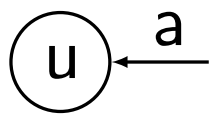
Алгоритм Aho-Corasick, создание суффиксных ссылок



$b = b + \varepsilon$

$bc = b + c$

Определение. l(v)=



l(u)=a

$l(r)=\varepsilon$

Определение. w(v)=

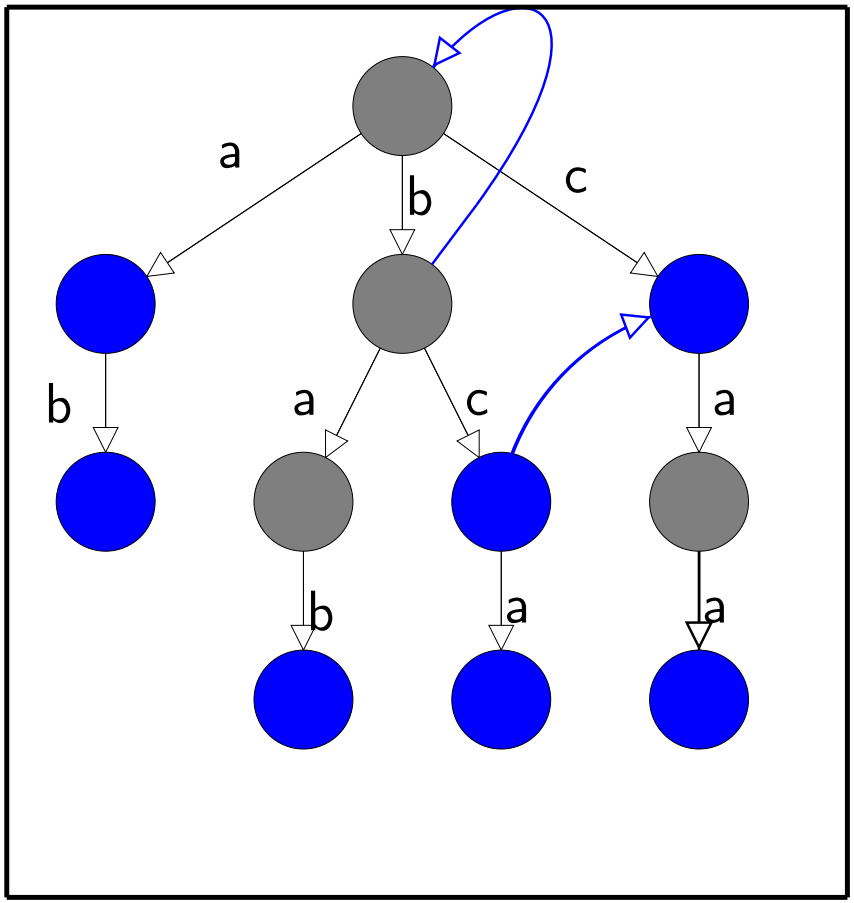
$r = v_0 \to v_1 \to \ldots \to v_n = v$

$w(v)=l(v_1)..l(v_n)$

Определение.

Суффиксная ссылка для u — это (u,v), где $w(u)=s+w(v)$, и $w(v)$ самый длинный суффикс $w(u)$. $u \neq v$.

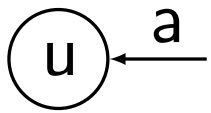
Алгоритм Aho-Corasick, создание суффиксных ссылок



$b = b + \varepsilon$

$bc = b + c$

Определение. l(v)=



l(u)=a

$l(r)=\varepsilon$

Определение. w(v)=

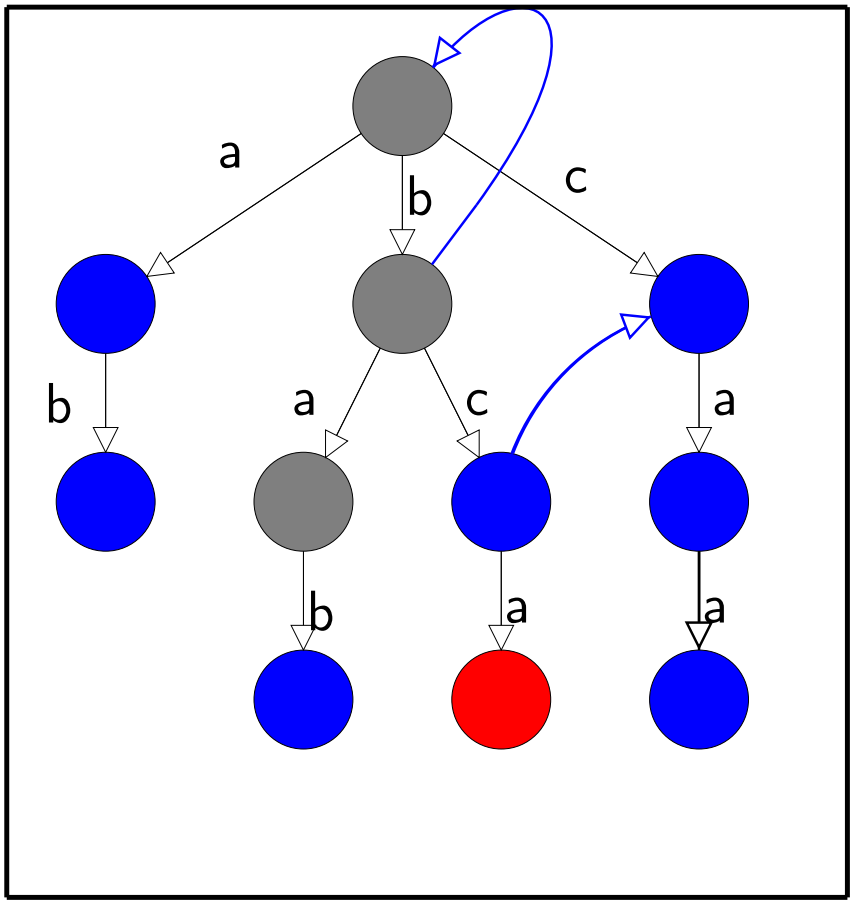
$r = v_0 \to v_1 \to \ldots \to v_n = v$

$w(v)=l(v_1)..l(v_n)$

Определение.

Суффиксная ссылка для u — это (u,v), где $w(u)=s+w(v)$, и $w(v)$ самый длинный суффикс $w(u)$. $u \neq v$.

Алгоритм Aho-Corasick, создание суффиксных ссылок

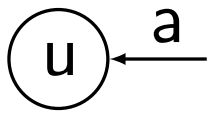


b = b + ε

bc = b + c

bca=b + ca

Определение. l(v)=



l(u)=a

l(r)=ε

Определение. w(v)=

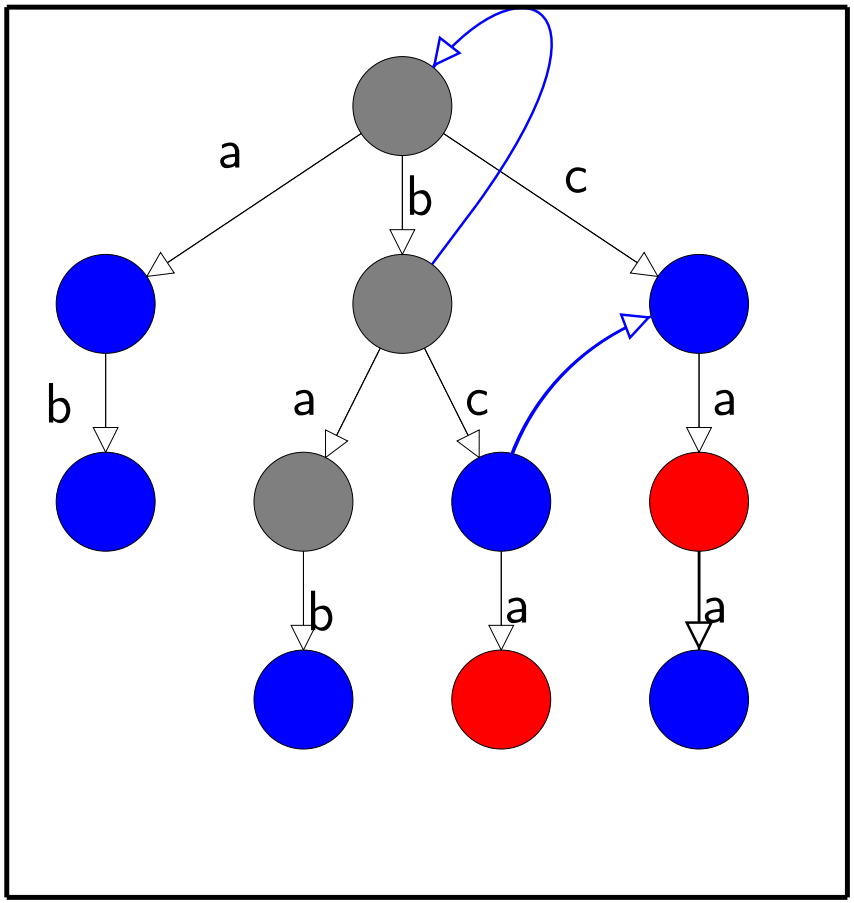
$r = v_0 \to v_1 \to \ldots \to v_n = v$

w(v)=l(v_1)..l(v_n)

Определение.

Суффиксная ссылка для u — это (u,v), где w(u)=s+w(v), и w(v) самый длинный суффикс w(u). $u \neq v$.

Алгоритм Aho-Corasick, создание суффиксных ссылок

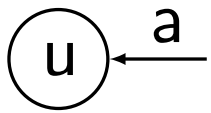


b = b + ε

bc = b + c

bca=b + ca

Определение. l(v)=



l(u)=a

l(r)=ε

Определение. w(v)=

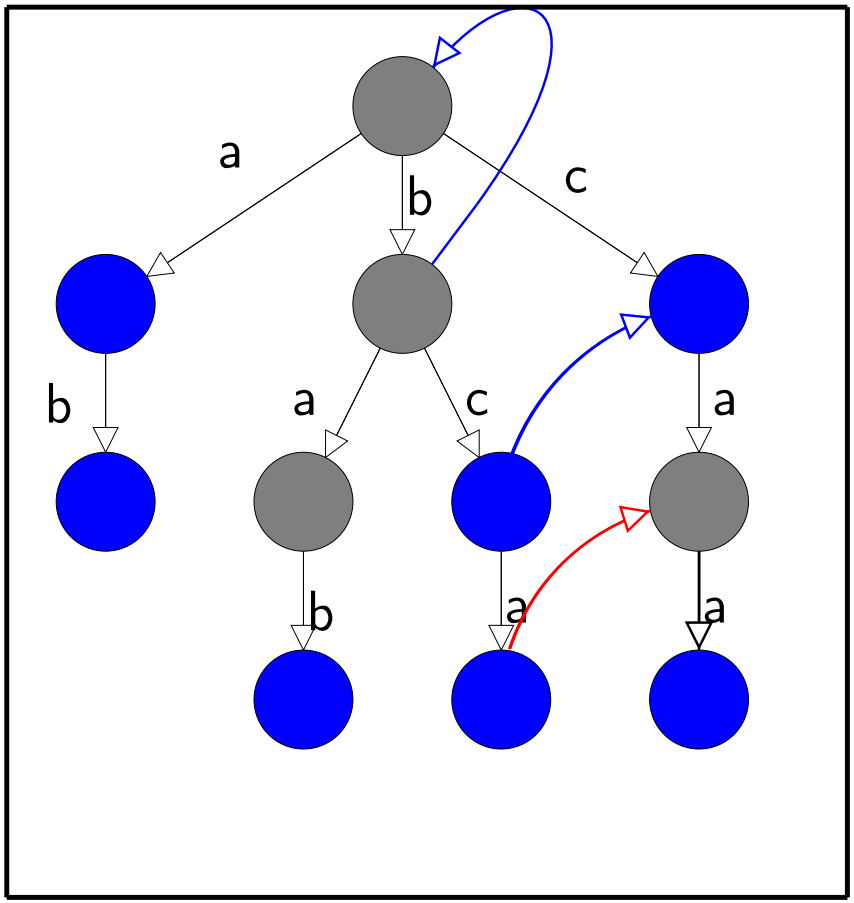
$r = v_0 \to v_1 \to \ldots \to v_n = v$

w(v)=l(v_1)..l(v_n)

Определение.

Суффиксная ссылка для u — это (u,v), где w(u)=s+w(v), и w(v) самый длинный суффикс w(u). $u \neq v$.

Алгоритм Aho-Corasick, создание суффиксных ссылок

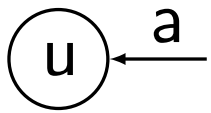


b = b + ε

bc = b + c

bca=b + ca

Определение. l(v)=



l(u)=a

l(r)=ε

Определение. w(v)=

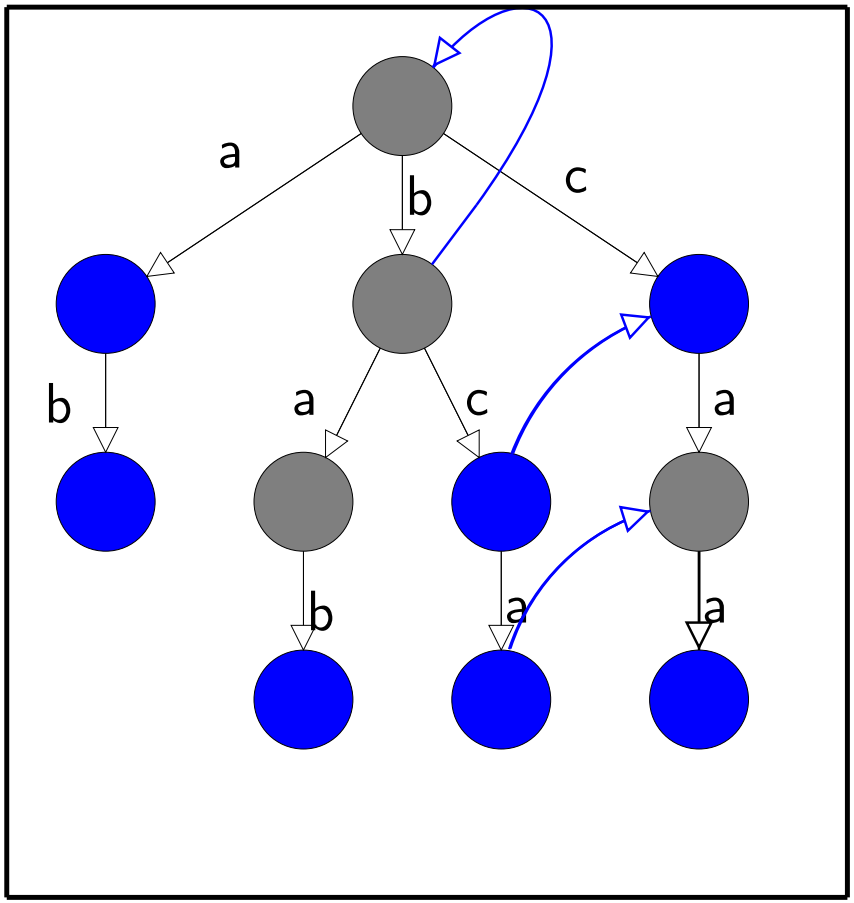
$r = v_0 \to v_1 \to \ldots \to v_n = v$

w(v)=l(v_1)..l(v_n)

Определение.

Суффиксная ссылка для u — это (u,v), где w(u)=s+w(v), и w(v) самый длинный суффикс w(u). $u \neq v$.

Алгоритм Aho-Corasick, создание суффиксных ссылок

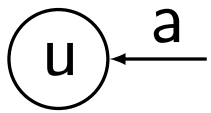


b = b + ε

bc = b + c

bca=b + ca

Определение. l(v)=



l(u)=a

l(r)=ε

Определение. w(v)=

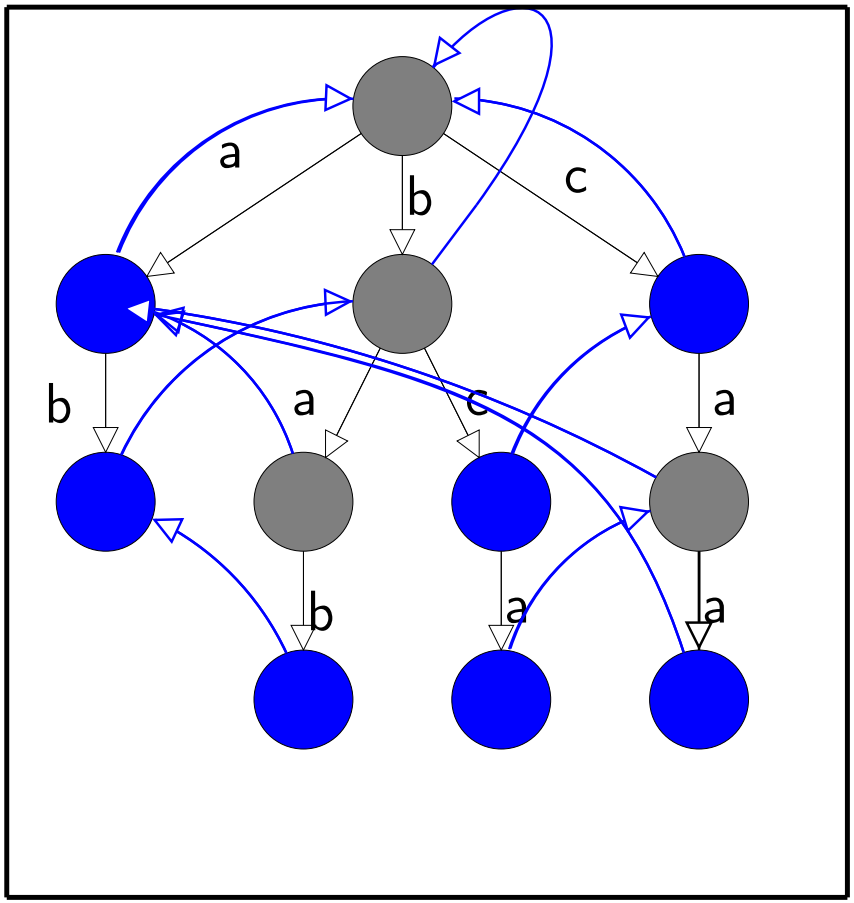
$r = v_0 \to v_1 \to \ldots \to v_n = v$

w(v)=l(v_1)..l(v_n)

Определение.

Суффиксная ссылка для u — это (u,v), где w(u)=s+w(v), и w(v) самый длинный суффикс w(u). $u \neq v$.

Алгоритм Aho-Corasick, создание суффиксных ссылок

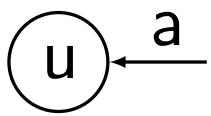


b = b + ε

bc = b + c

bca=b + ca

Определение. l(v)=



l(u)=a

l(r)=ε

Определение. w(v)=

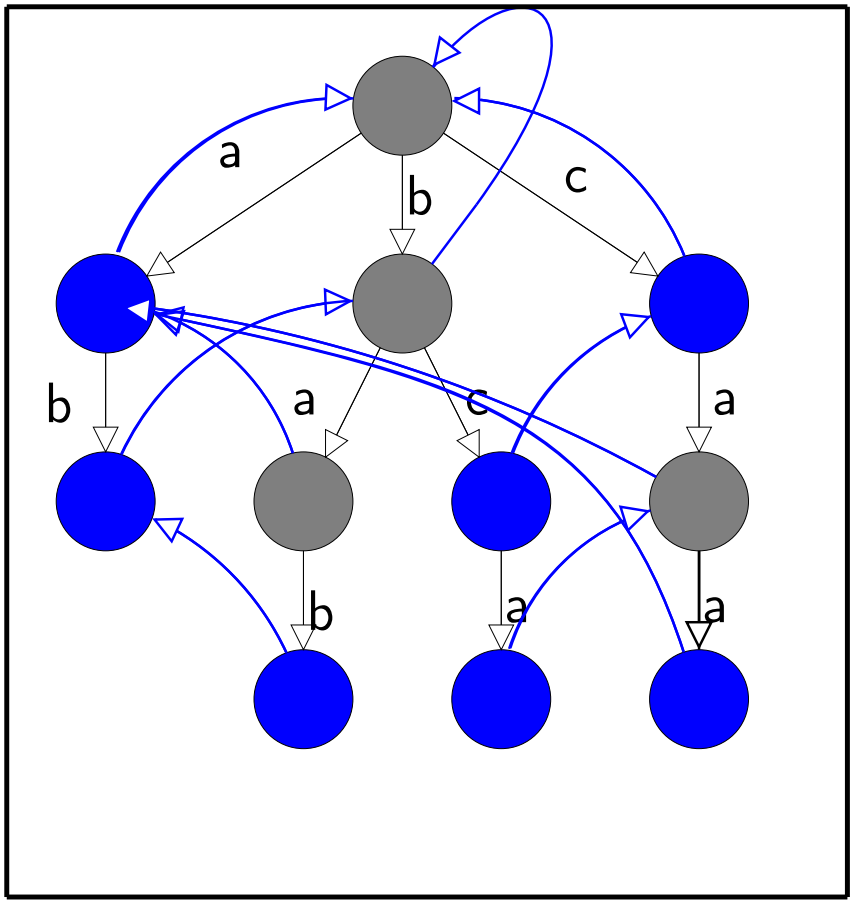
$r = v_0 \to v_1 \to \ldots \to v_n = v$

w(v)=l(v_1)..l(v_n)

Определение.

Суффиксная ссылка для u — это (u,v), где w(u)=s+w(v), и w(v) самый длинный суффикс w(u). $u \neq v$.

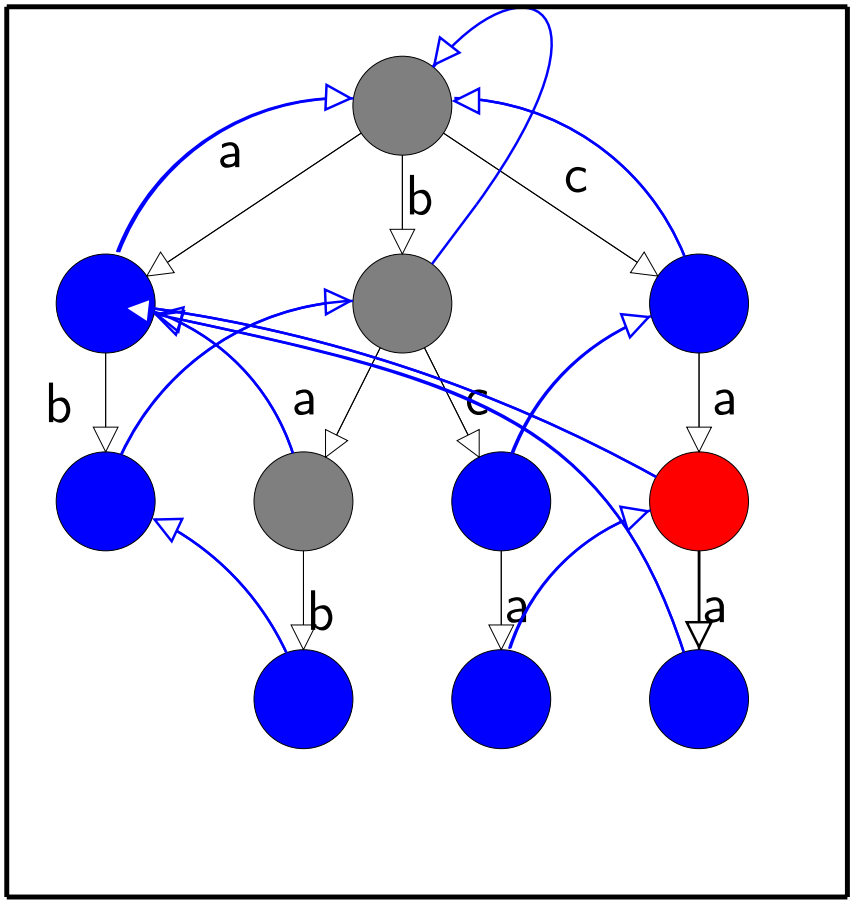
Алгоритм Aho-Corasick, создание словарных ссылок



{a, ab, bab, bc, bca, c, caa}

Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

Алгоритм Aho-Corasick, создание словарных ссылок

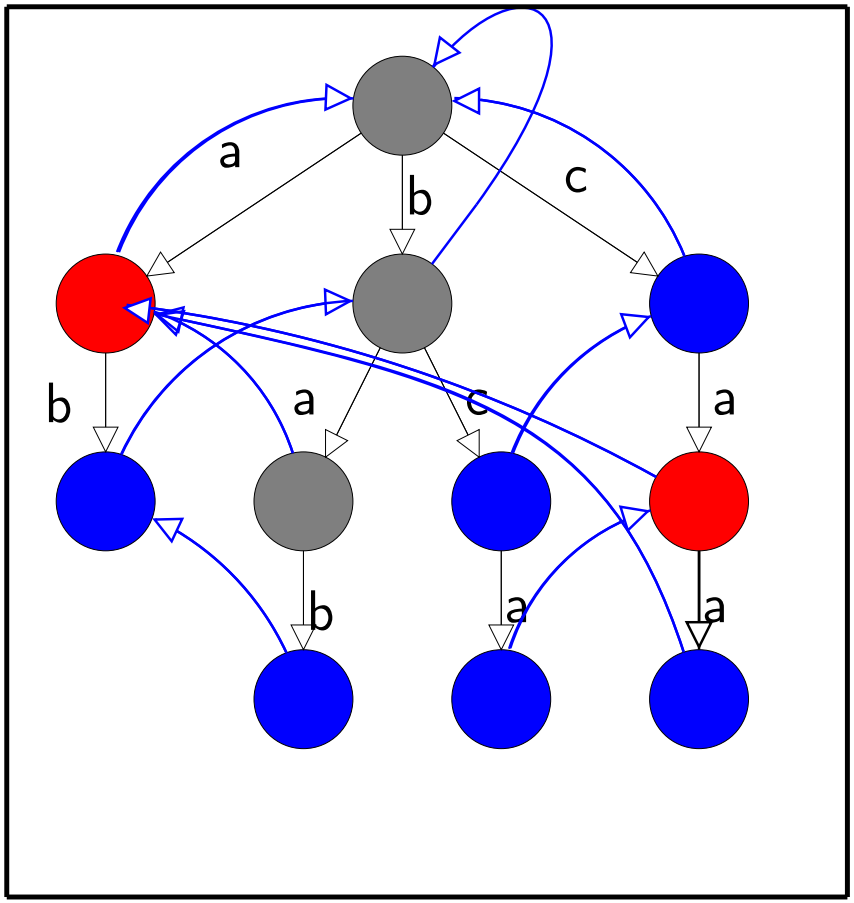


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a

Алгоритм Aho-Corasick, создание словарных ссылок

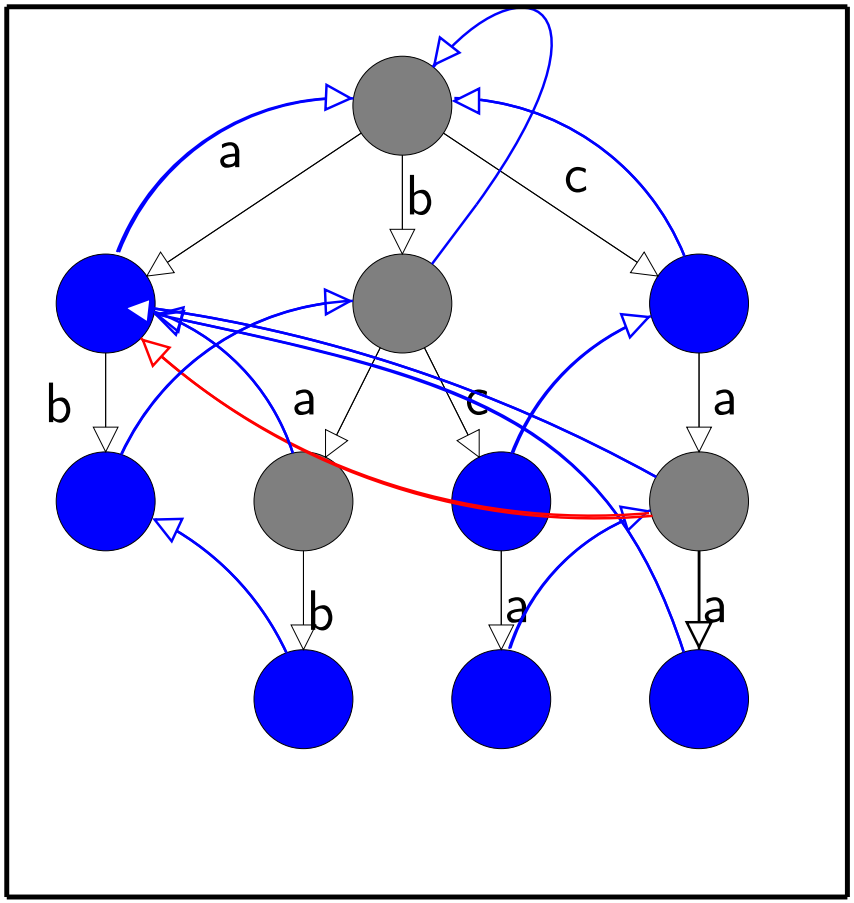


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a

Алгоритм Aho-Corasick, создание словарных ссылок

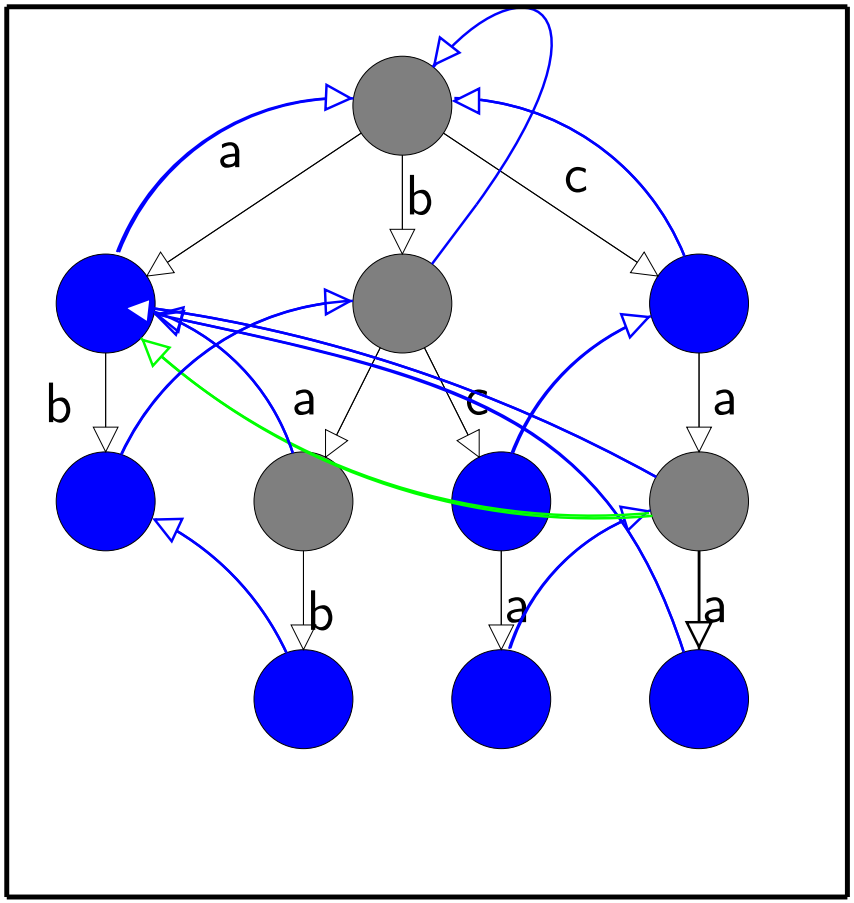


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a

Алгоритм Aho-Corasick, создание словарных ссылок

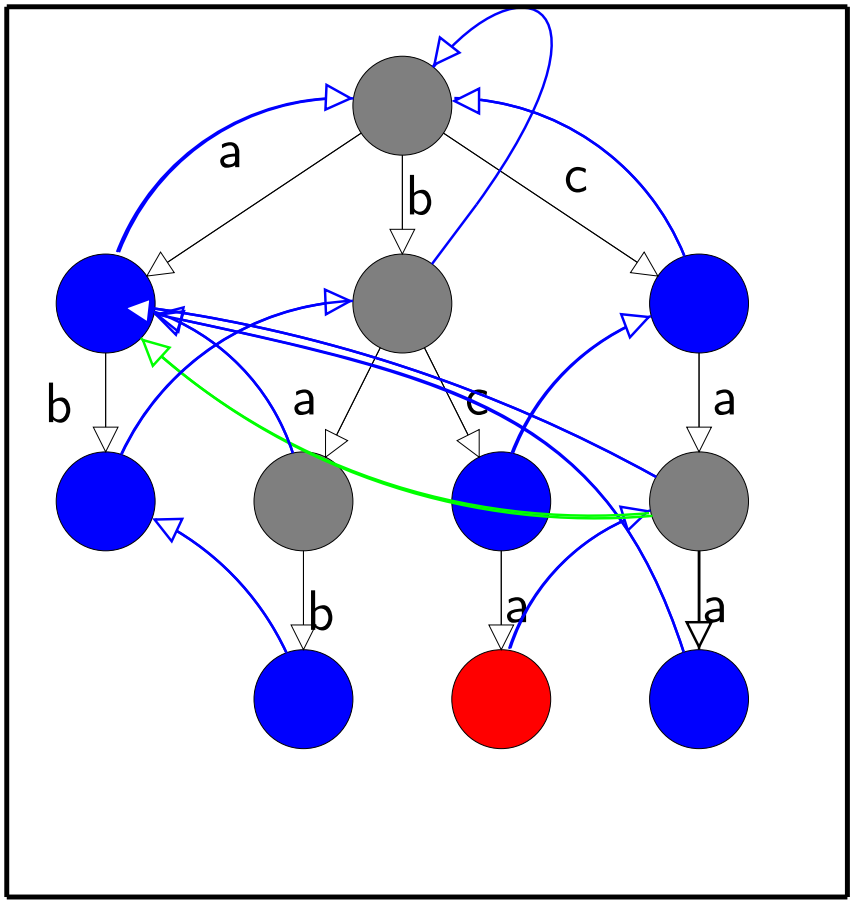


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a

Алгоритм Aho-Corasick, создание словарных ссылок

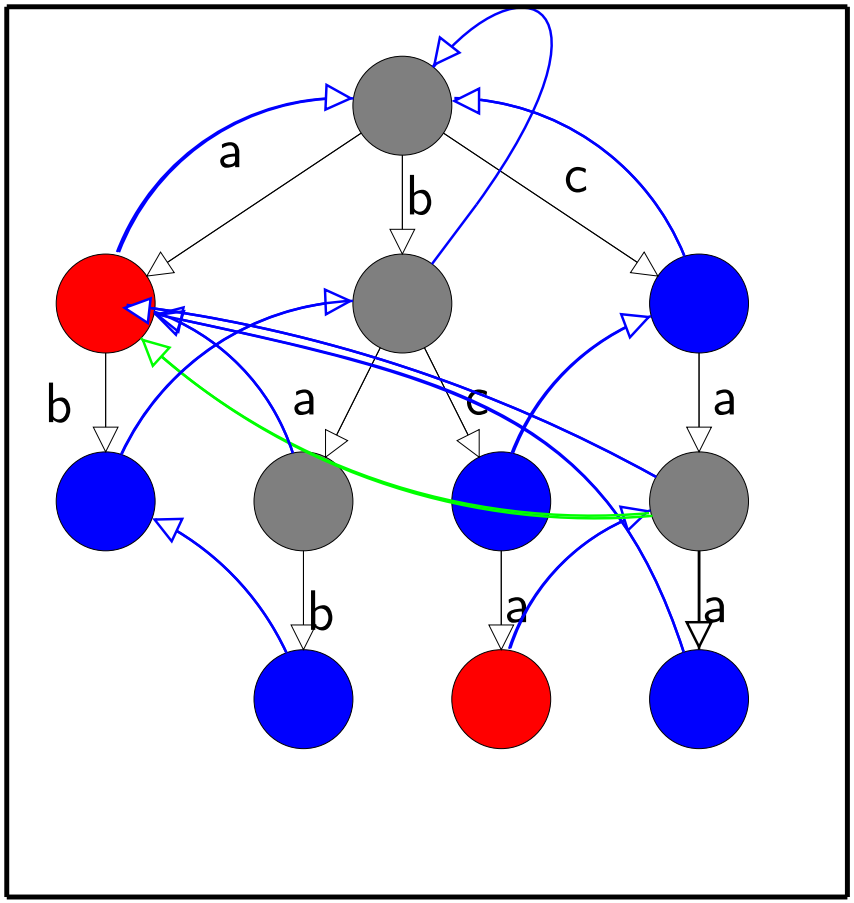


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a
bca=bc+a

Алгоритм Aho-Corasick, создание словарных ссылок

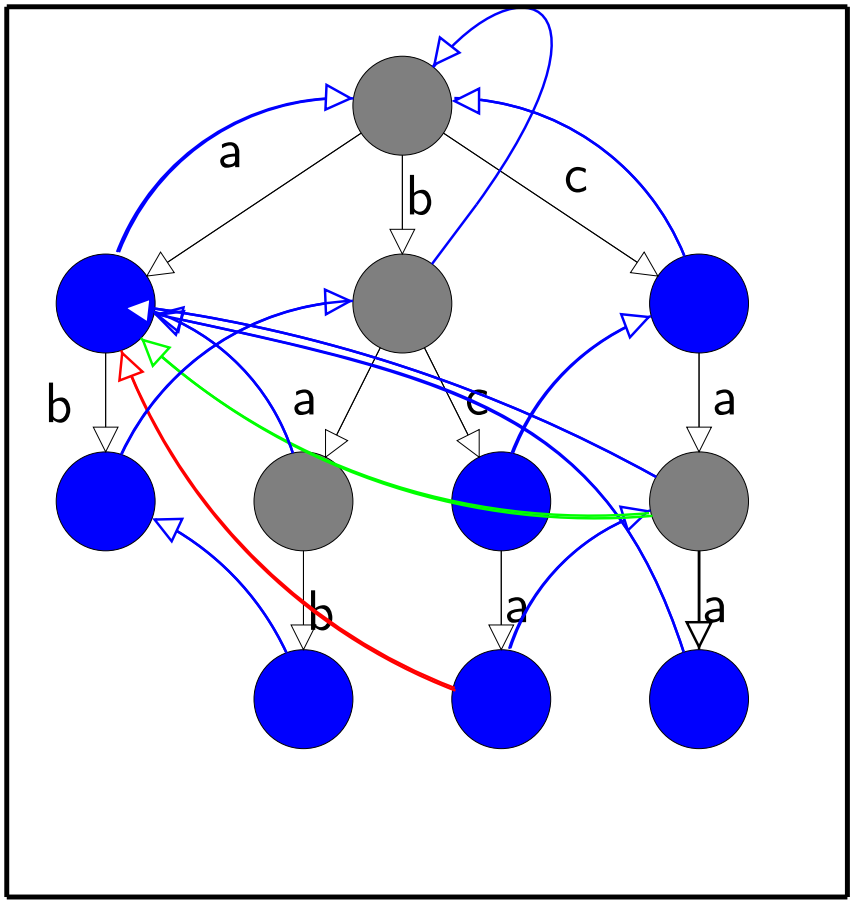


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a
bca=bc+a

Алгоритм Aho-Corasick, создание словарных ссылок

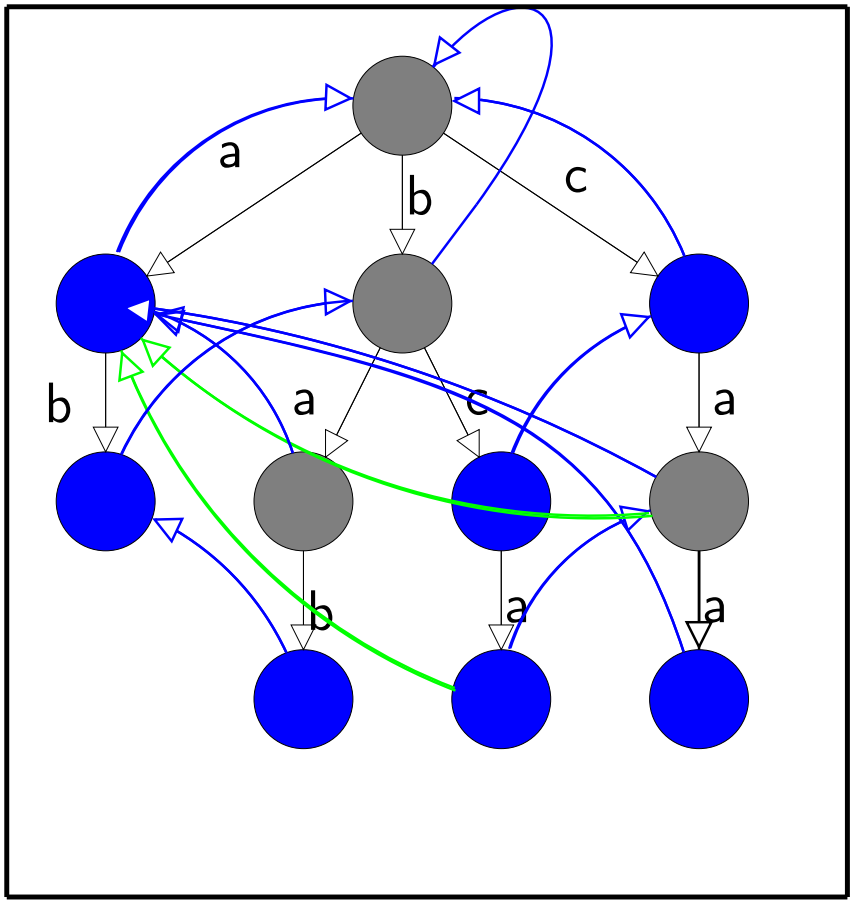


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a
bca=bc+a

Алгоритм Aho-Corasick, создание словарных ссылок

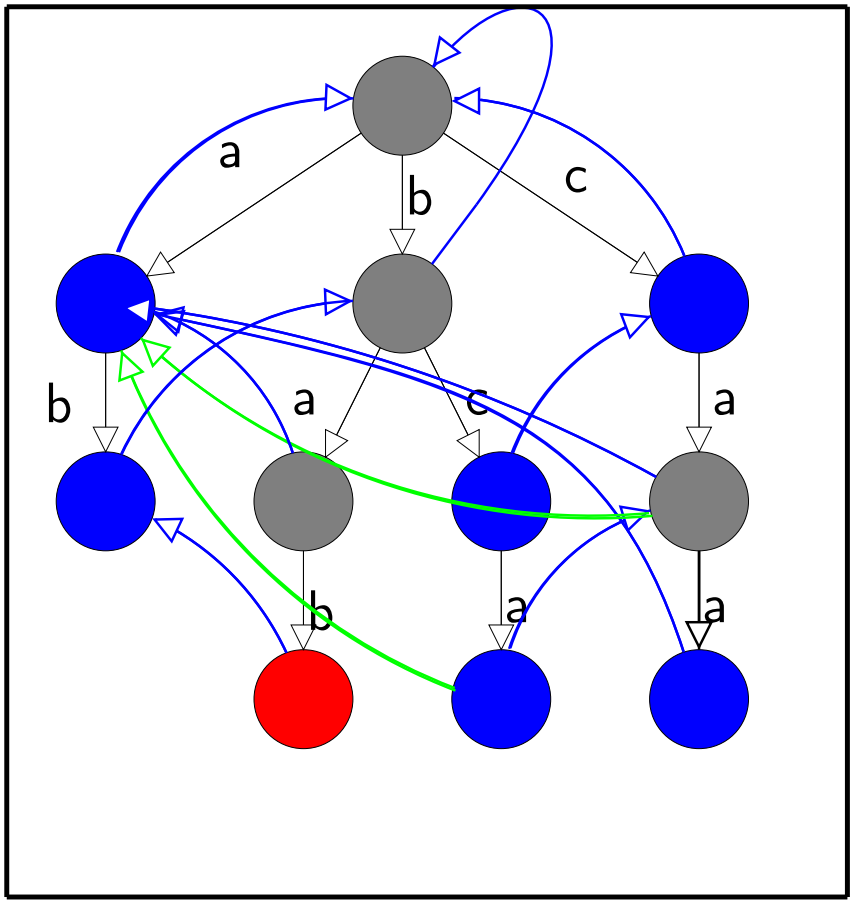


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a
bca=bc+a

Алгоритм Aho-Corasick, создание словарных ссылок

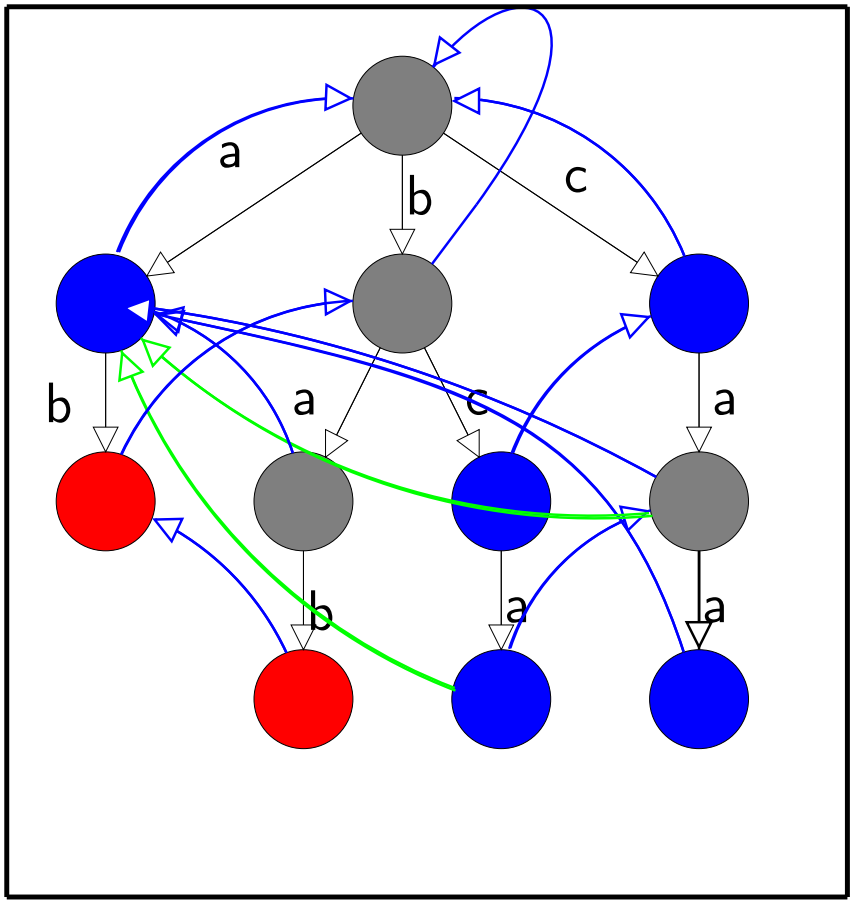


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a
bca=bc+a
bab=b+ab

Алгоритм Aho-Corasick, создание словарных ссылок

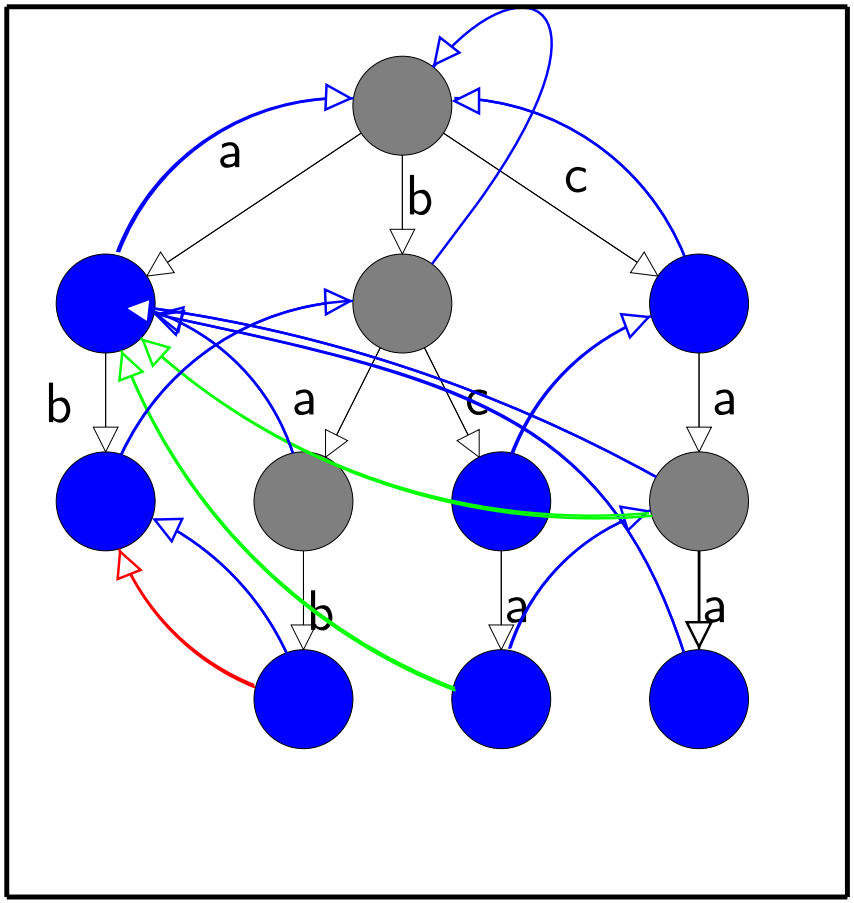


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a
bca=bc+a
bab=b+ab

Алгоритм Aho-Corasick, создание словарных ссылок

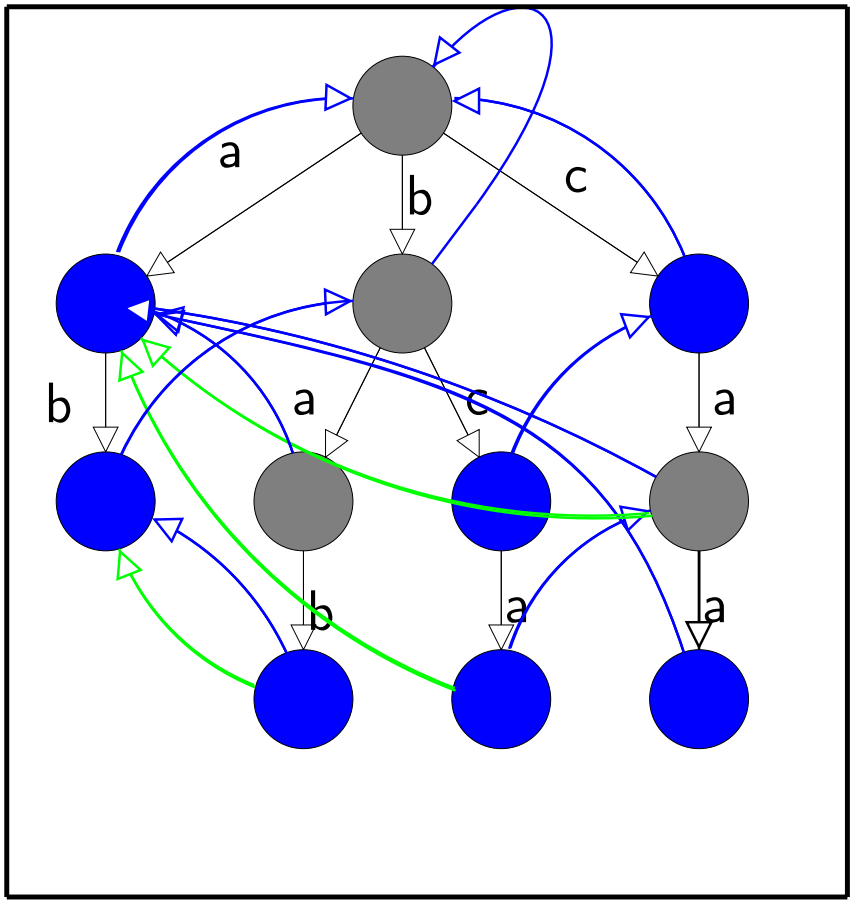


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a
bca=bc+a
bab=b+ab

Алгоритм Aho-Corasick, создание словарных ссылок

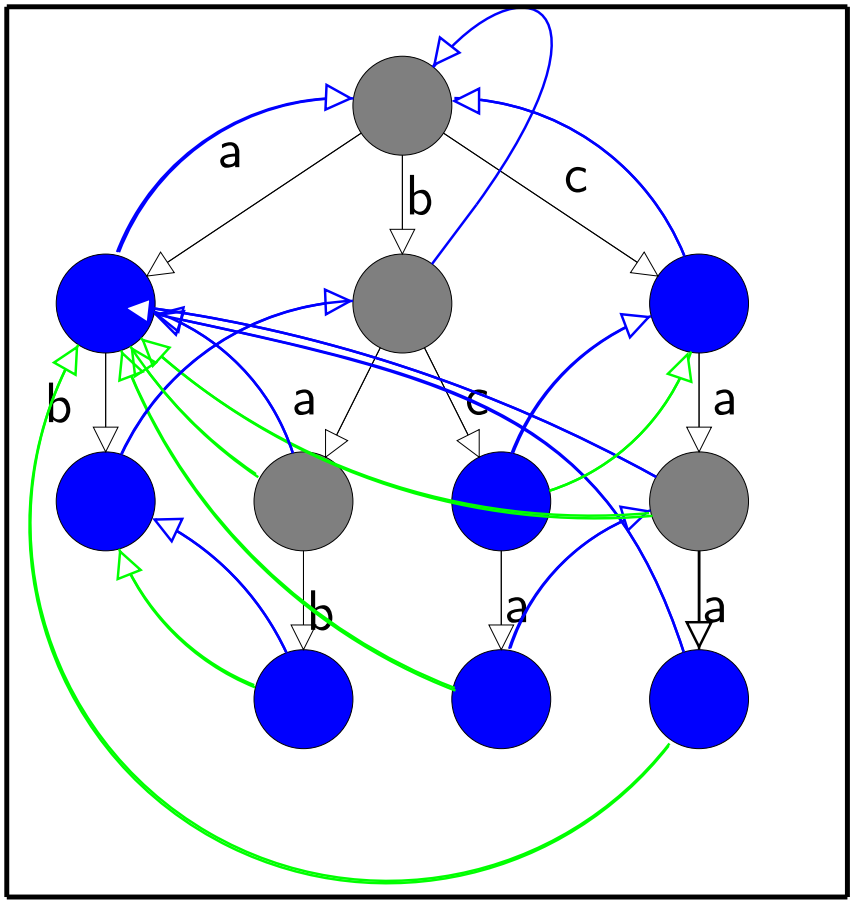


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a
bca=bc+a
bab=b+ab

Алгоритм Aho-Corasick, создание словарных ссылок

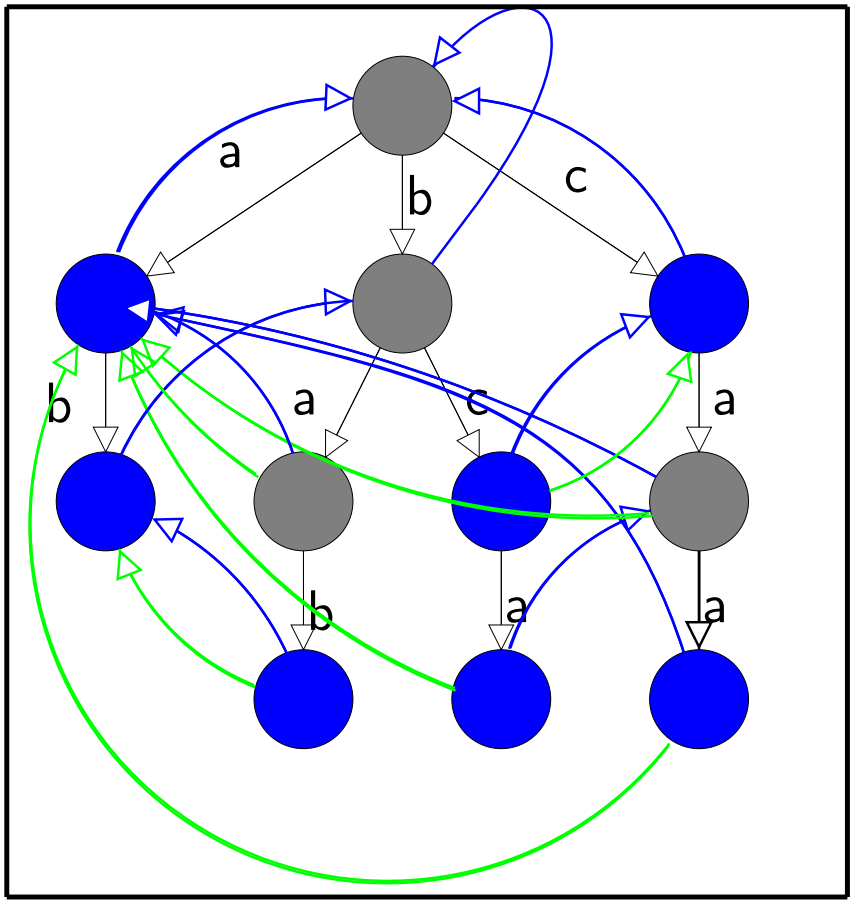


Определение. Пусть
$w(u) = s_1 + w(u_1)$
...
$w(u) = s_t + w(u_t)$,
где $w(u_1), .., w(u_t)$ – шаблоны.
Тогда $(u, u_1), .., (u, u_t)$ – словарные ссылки

{a, ab, bab, bc, bca, c, caa}

ca=c+a
bca=bc+a
bab=b+ab

Алгоритм Aho-Corasick, поиск.

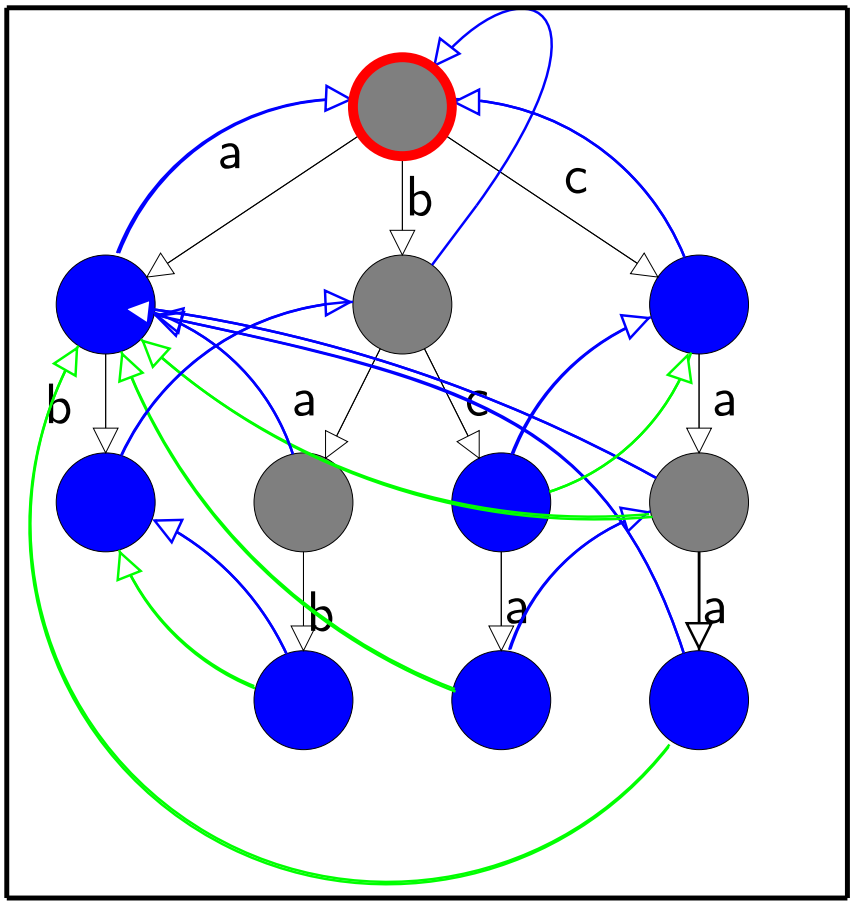


{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

Алгоритм Aho-Corasick, поиск.

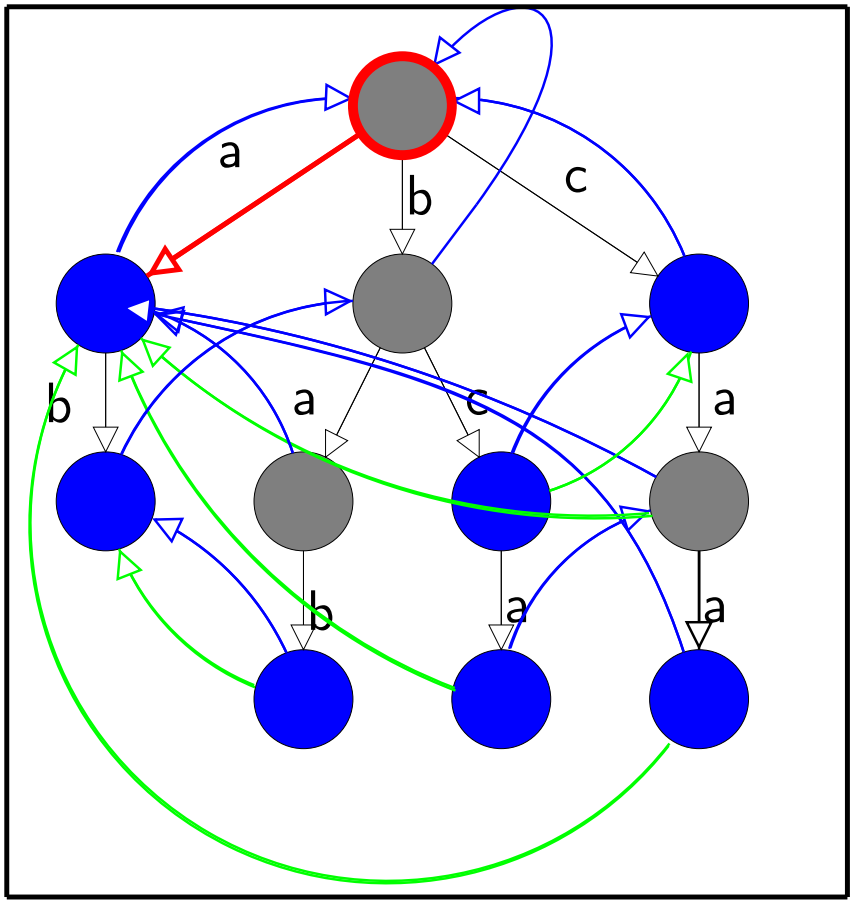


{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

Алгоритм Aho-Corasick, поиск.

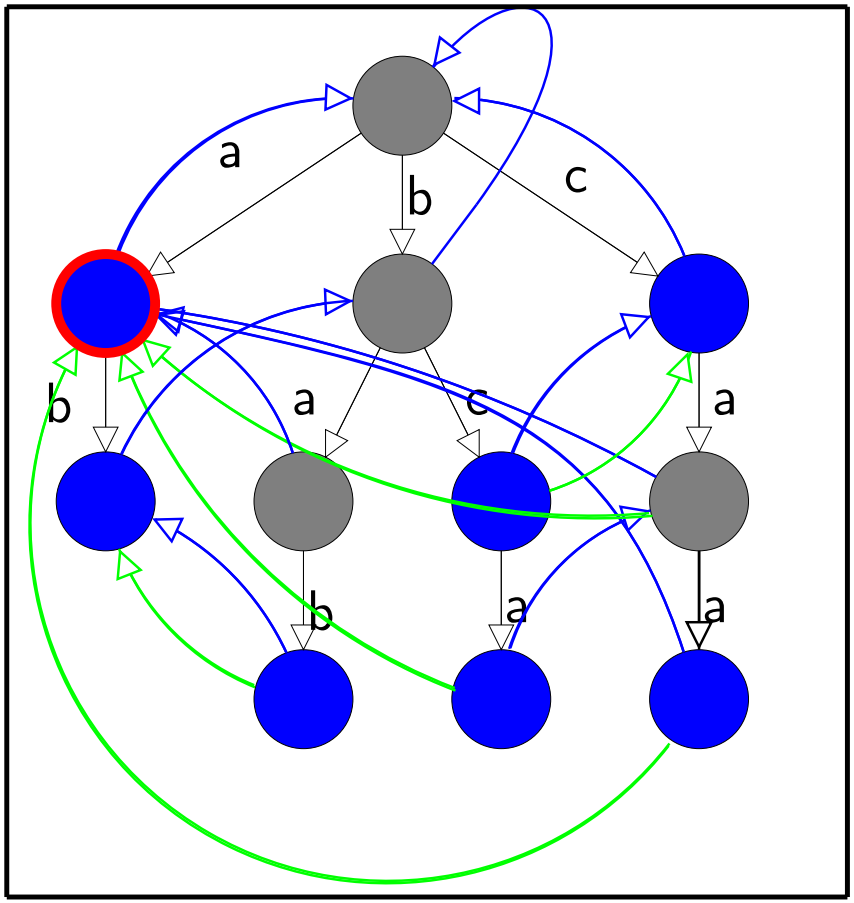


{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

Алгоритм Aho-Corasick, поиск.



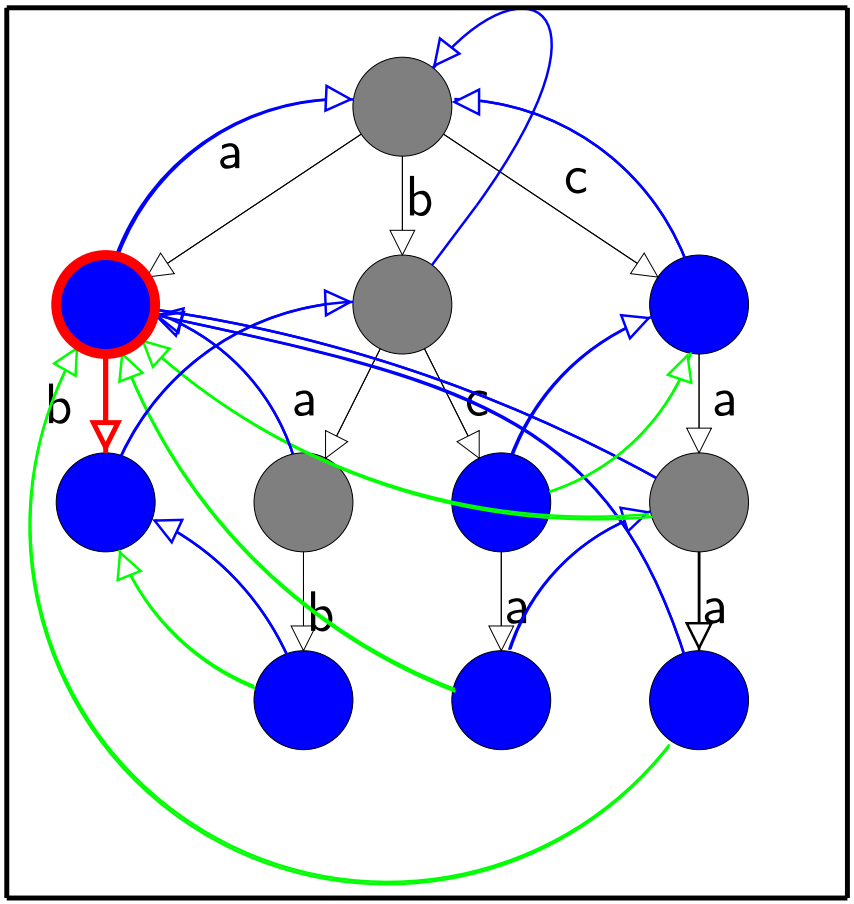
{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

a

Алгоритм Aho-Corasick, поиск.



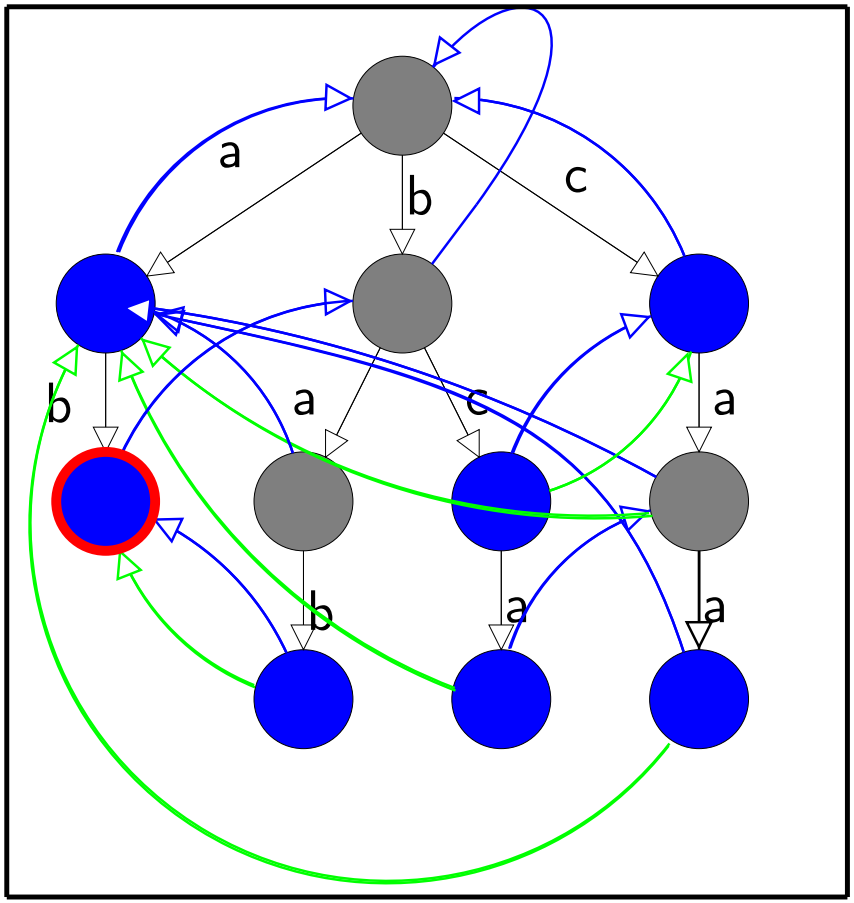
{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

a

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

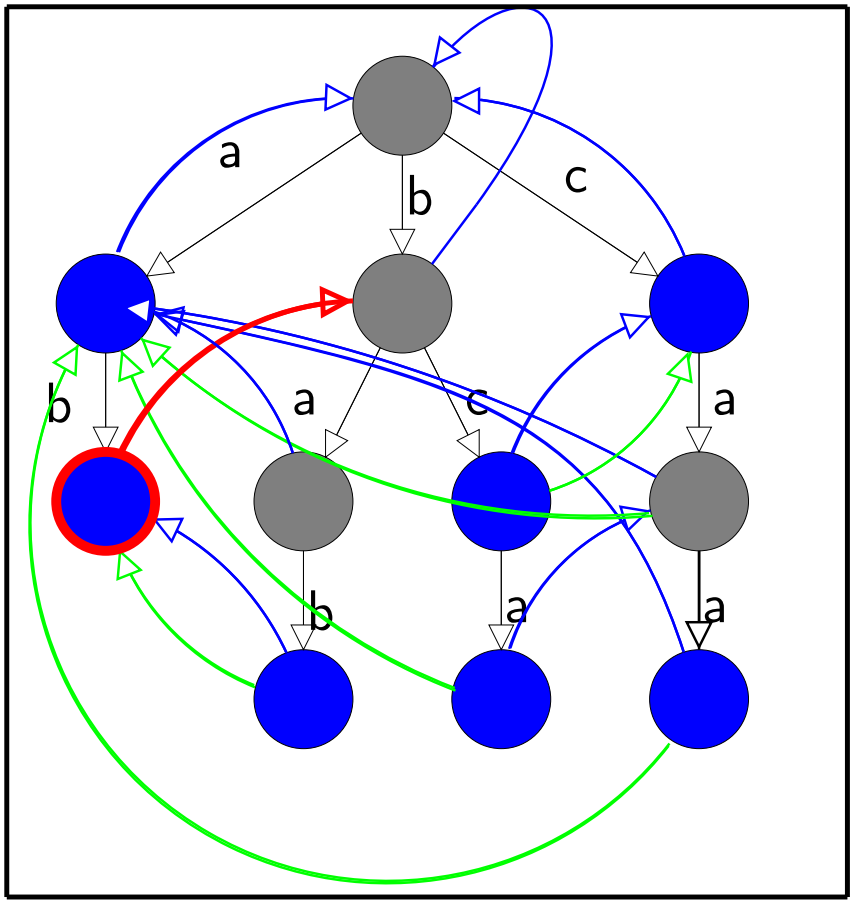
TEXT = abccab

Вхождение:

a

ab

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

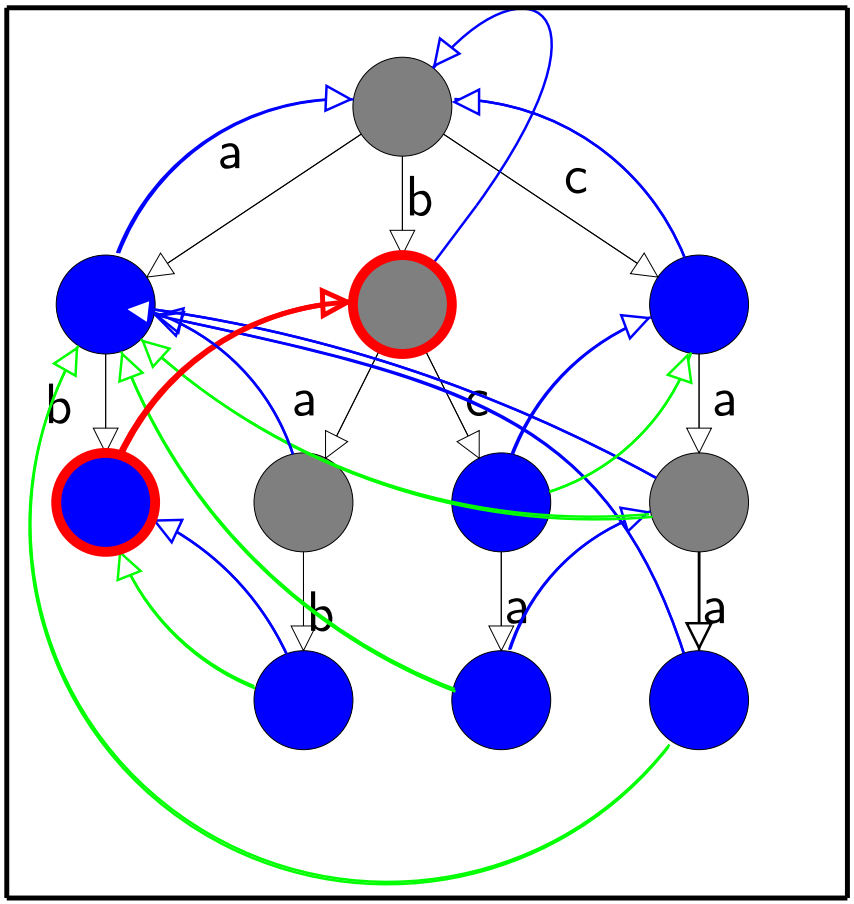
TEXT = abccab

Вхождение:

a

ab

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

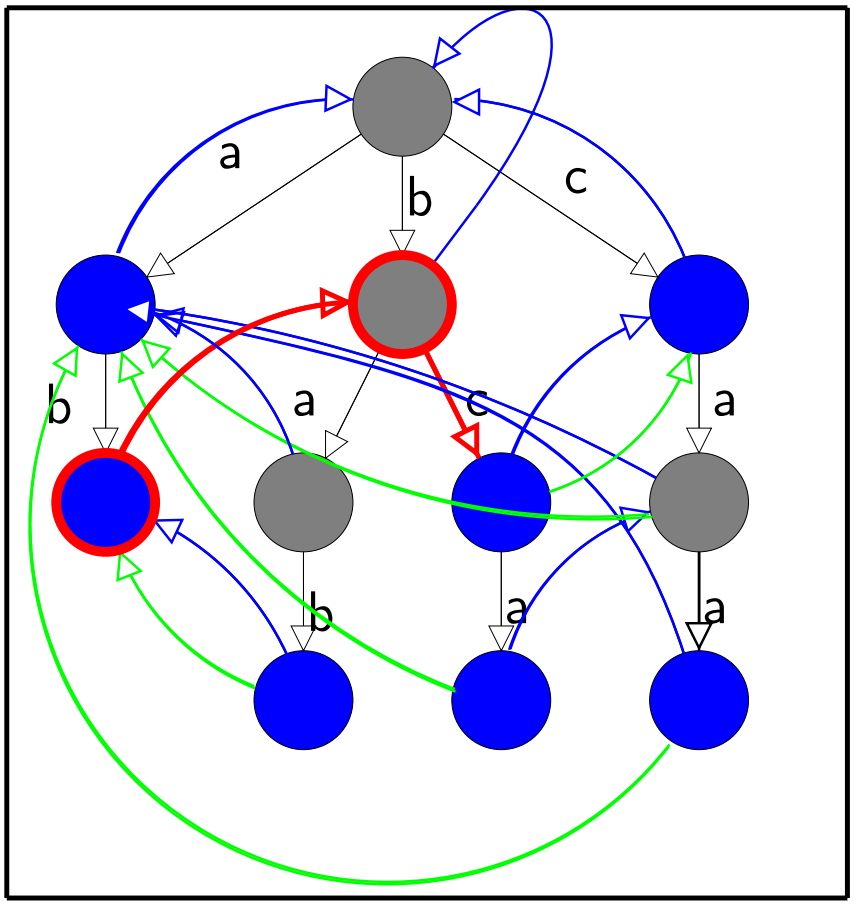
TEXT = abccab

Вхождение:

a

ab

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

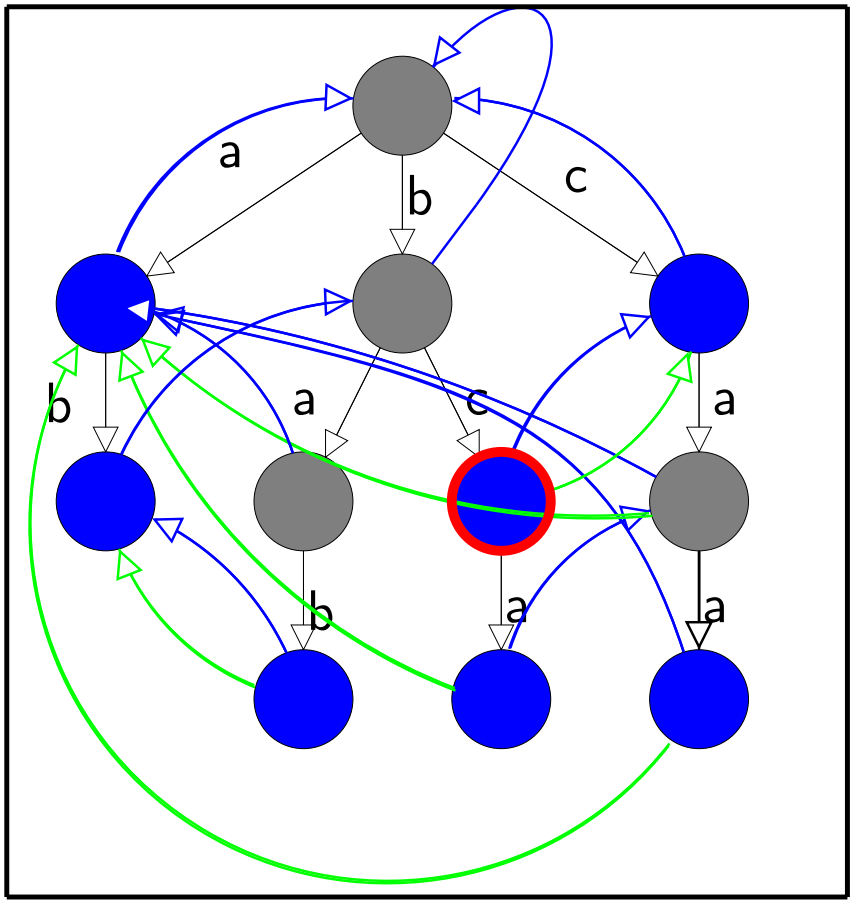
TEXT = abccab

Вхождение:

a

ab

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

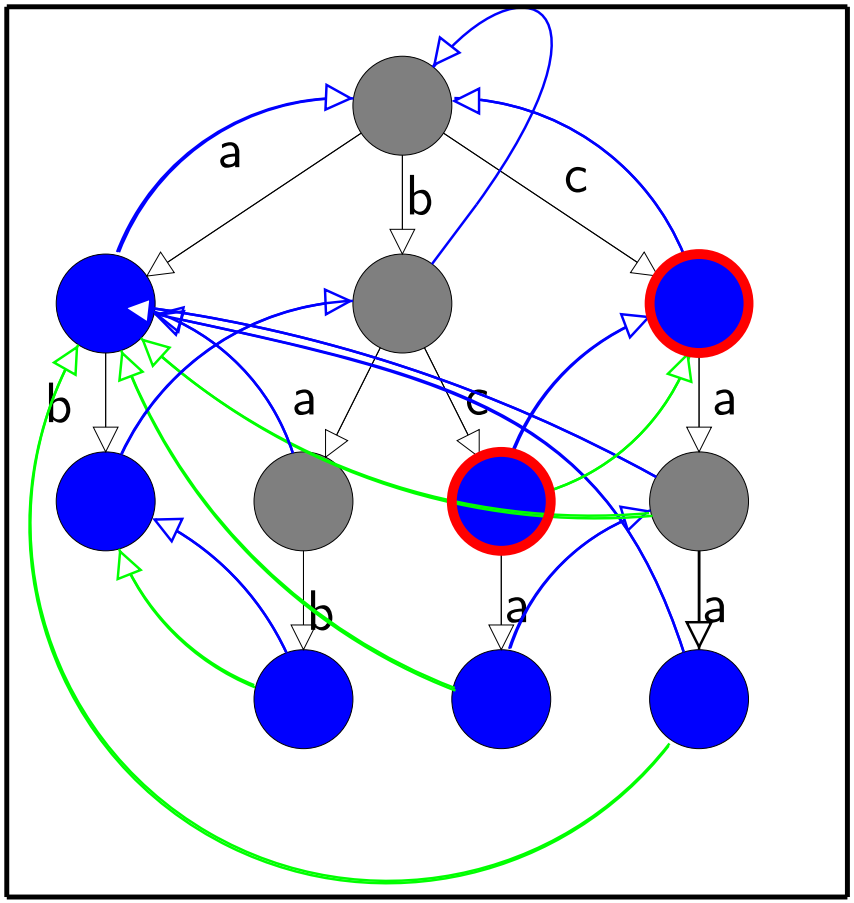
Вхождение:

a

ab

bc

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

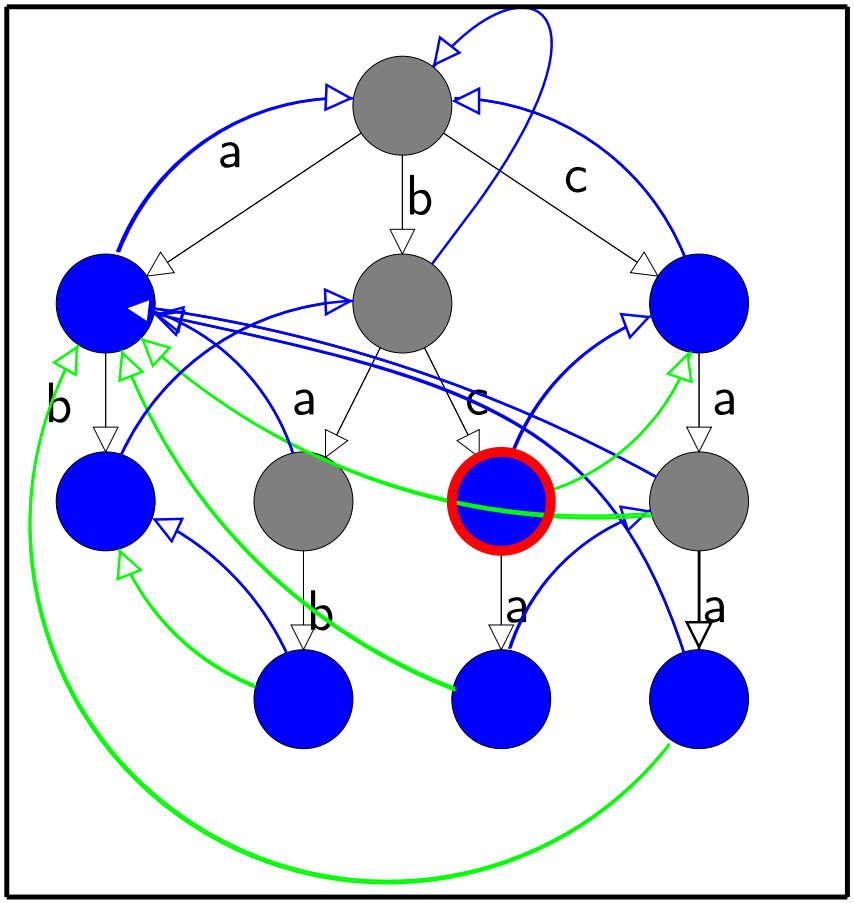
Вхождение:

a

ab

bc,c

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

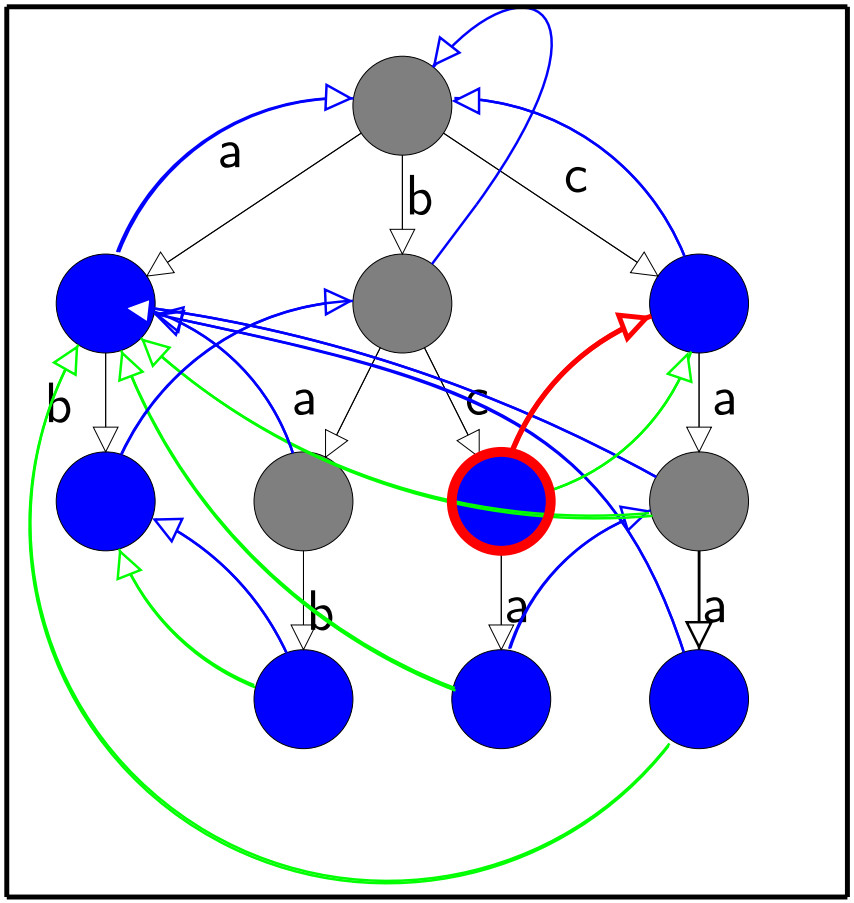
Вхождение:

a

ab

bc,c

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

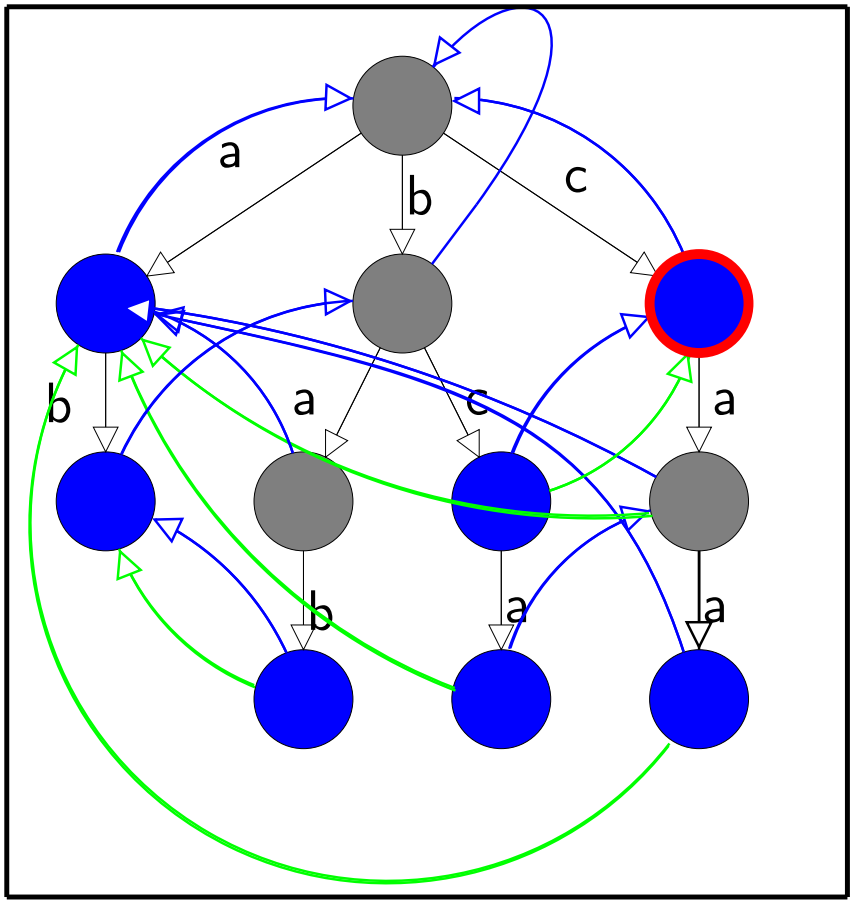
Вхождение:

a

ab

bc,c

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

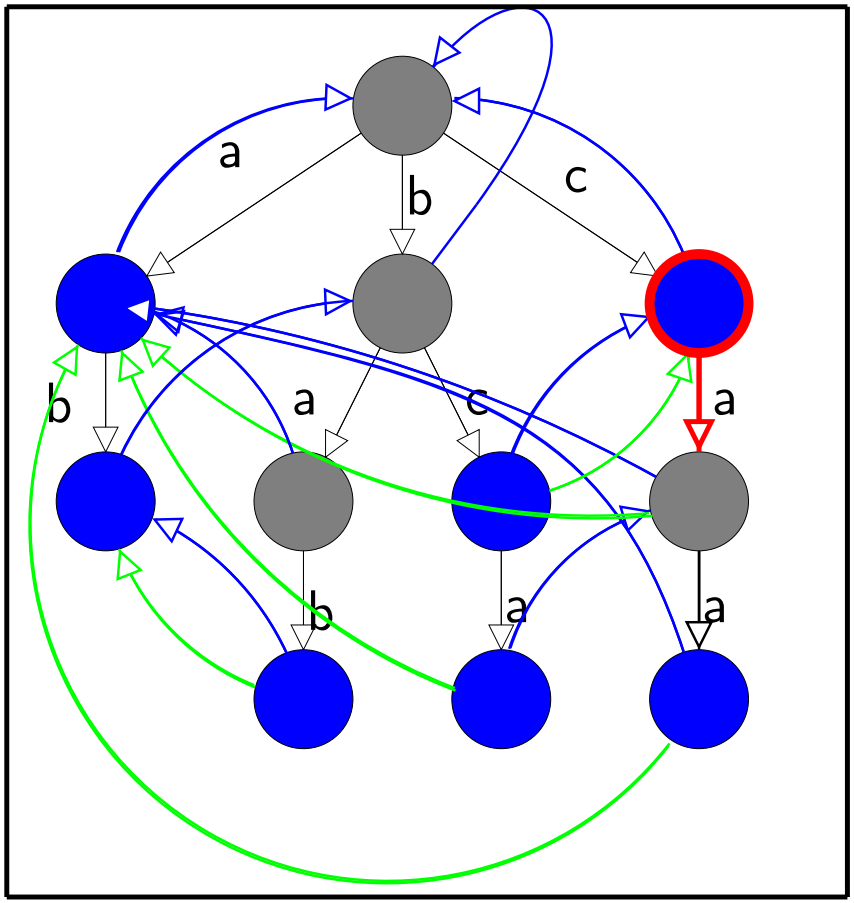
Вхождение:

a

ab

bc,c

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

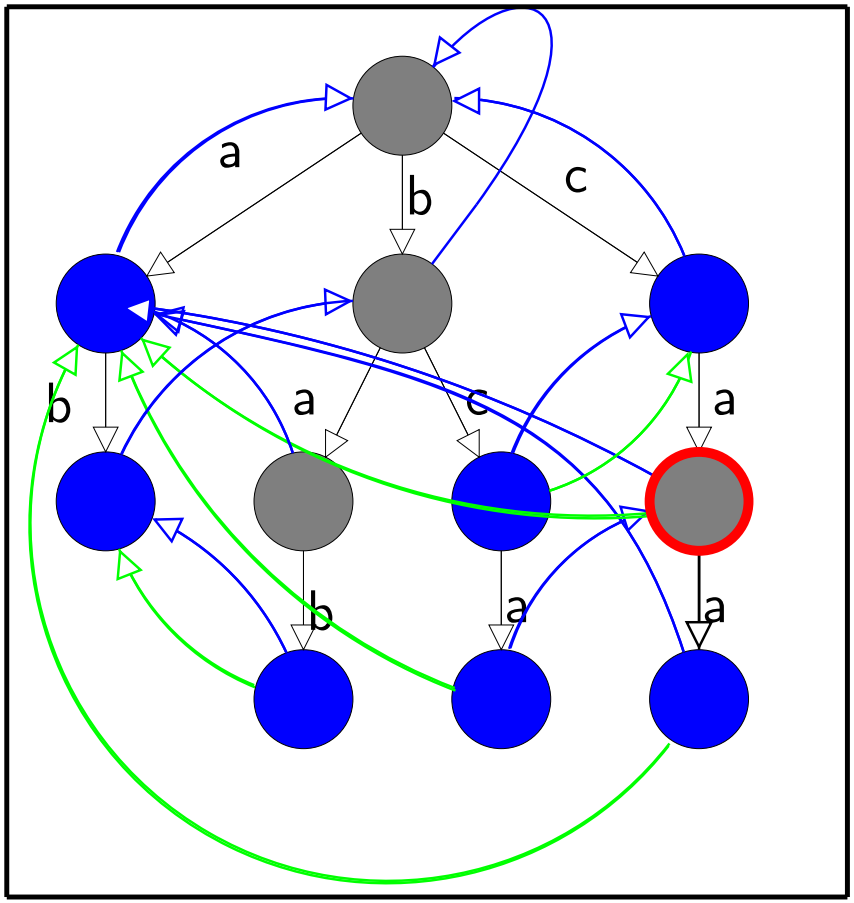
Вхождение:

a

ab

bc,c

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

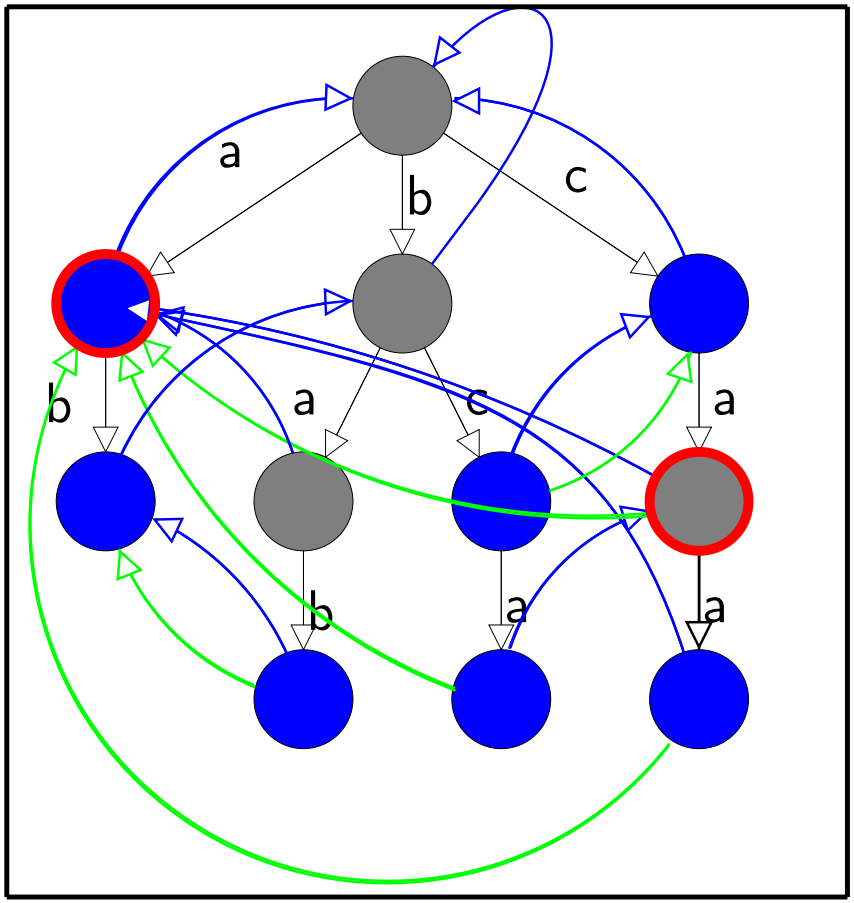
Вхождение:

a

ab

bc,c

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

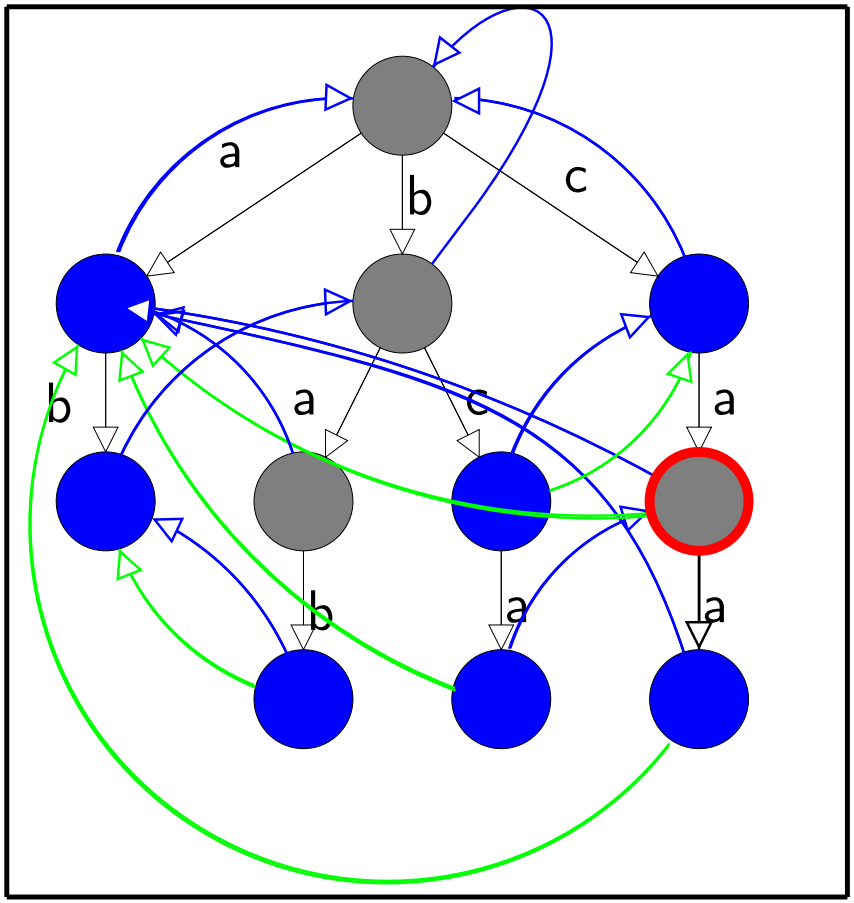
a

ab

bc,c

a

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

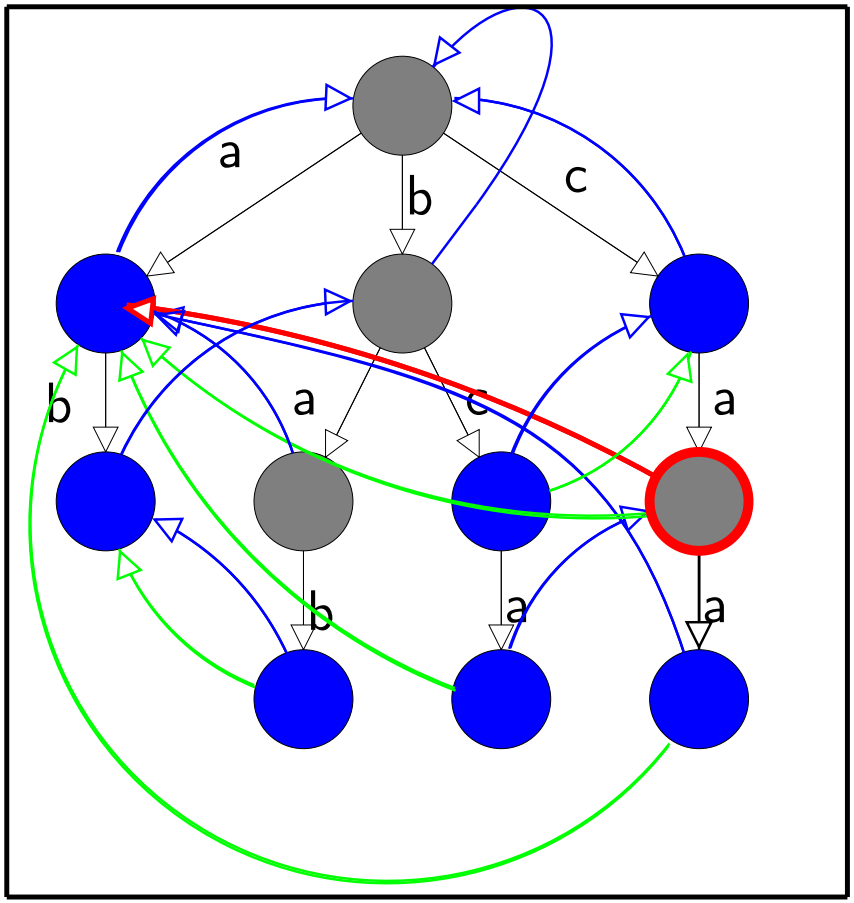
a

ab

bc,c

a

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

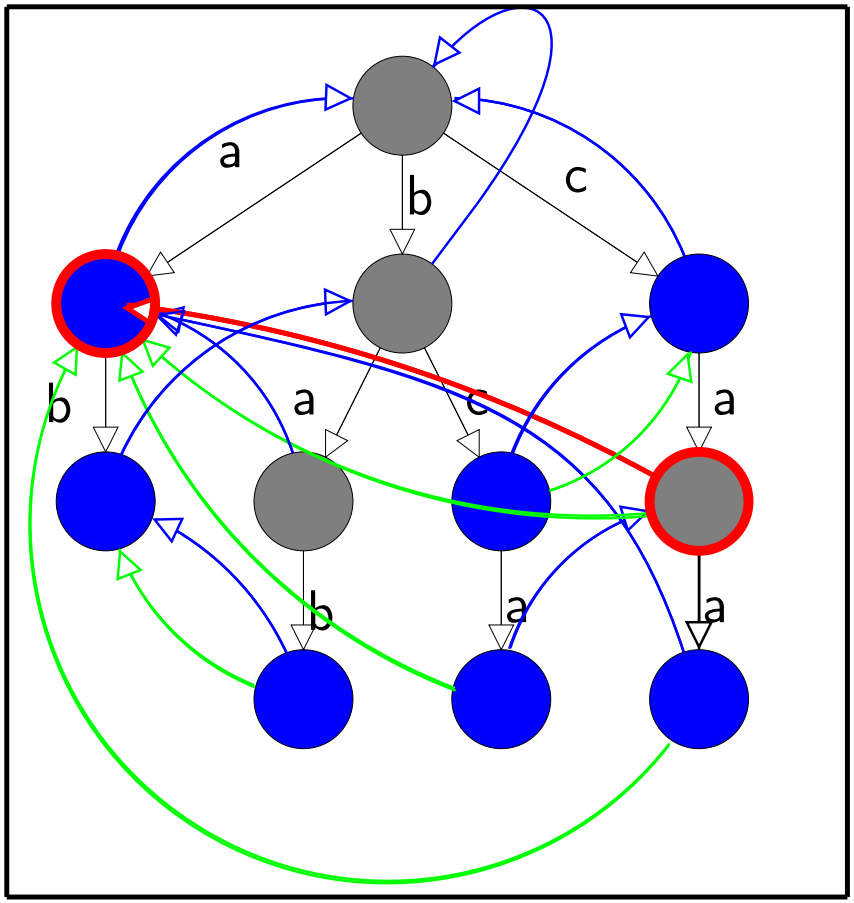
a

ab

bc,c

a

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

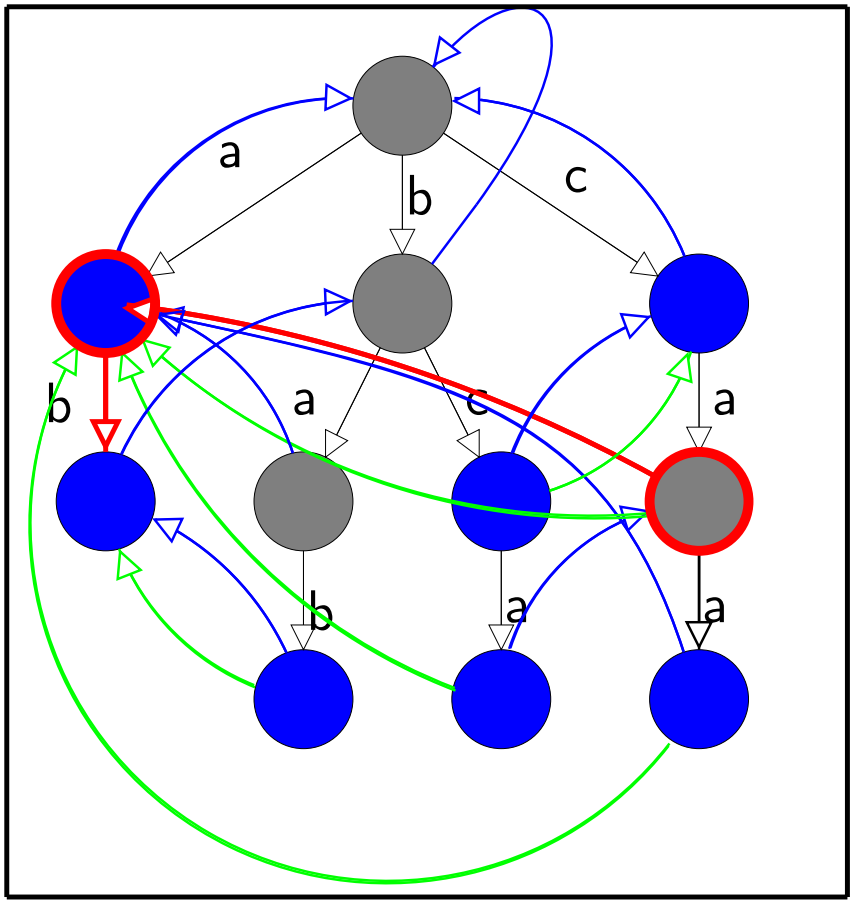
a

ab

bc,c

a

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

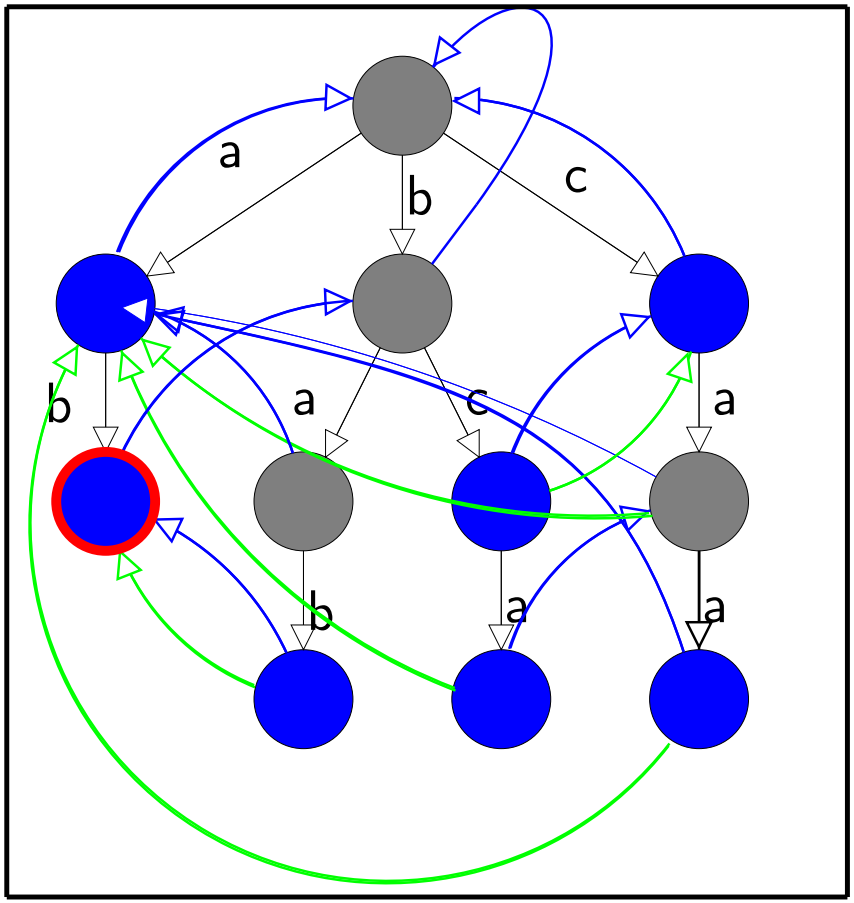
a

ab

bc,c

a

Алгоритм Aho-Corasick, поиск.



{a, ab, bab, bc, bca, c, caa}

TEXT = abccab

Вхождение:

a

ab

bc,c

a

ab

Классические подходы. Общая идея алгоритма Rabin-Karp

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

Пусть
subsP=$\{P_1..P_n\}$ – шаблоны.
$l_1 <..< l_s$ – длины шаблонов
h – хэш-функция.
T – текст

Классические подходы. Общая идея алгоритма Rabin-Karp

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

Пусть множество hashes есть $\{h(P_1),\ldots,h(P_n)\}$

Классические подходы. Общая идея алгоритма Rabin-Karp

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

Первым делом начальный отрезок строки длины $l_s - 1$ проверяется на наличие шаблонов, чтобы не вылезти за границу массива основным алгоритмом.

Классические подходы. Общая идея алгоритма Rabin-Karp

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

Далее идет основной цикл.

Классические подходы. Общая идея алгоритма Rabin-Karp

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

Вначале в цикле вычисляются хэши $h_1..h_s$ от подстрок текста. На каждой итерации нужно вычислять хэш-функцию от подстрок с длинами $l_1,..,l_s$

Классические подходы. Общая идея алгоритма Rabin-Karp

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

Далее для каждого h_i проверяется принадлежность к hashes.

Классические подходы. Общая идея алгоритма Rabin-Karp

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

Такая проверка требует меньше затрат, чем проверка принадлежности подстроки к subsP.
Однако из того, что $h_i \in$ hashes $\not\Rightarrow$ T[j-l_i+1,..,j] $\in$ subsP

Классические подходы. Общая идея алгоритма Rabin-Karp

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

В случае, если $h_i \in$ hashes производится проверка $T[j-l_i+1,..,j] \in subsP$

Классические подходы. Общая идея алгоритма Rabin-Karp

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

Если проверка сработаола, то найдено вхождение шаблона в текст.

Реализации. GNU grep.

GNU grep может искать множественные шаблоны в двух режимах.

- Множественные фиксированные шаблоны(не регулярные) ищутся при помощи алгоритма Commentz-Walter.
- Множественные регулярные выражения ищутся при помощи DFA.

CUDA-grep

Существует паттерн-матчер CUDA-grep. Эта программа, которая строит NFA алгоритмом Томпсона. И при помощи этого автомата находит регулярные выражения в тексте. Эта программа реализована на CUDA.

Эта программа не работала на CUDA 5 из-за ошибок реализации. Мы выслали патч разработчикам, который исправляет эти ошибки. (см. pull requests `https://github.com/bkase/CUDA-grep`)

Однако сравниться на поиске множества шаблонов, с ним не удалось. Если подать CUDA-grep длинное регулярное выражение $\mathtt{P}_1|\ldots|\mathtt{P}_n$, то в нем наступит переполнение массивов и программа аварийно завершится.

Реализации. CUDA-grep. pull request.

```
180 184
181 185                         endCopyStringsToDevice = CycleTimer::currentSeconds();
182 186
183     -                       u32 numRegexes = 1;
184     -                       pMatch(device_line, device_line_table, num_lines, 1, timerOn, device_regex, &numRegexes,
    187 +                       u32 host_regex_table[1]; /*offsets to regexes on host*/
    188 +                       u32 *device_regex_table; /*this array will contain host_regex_table*/
    189 +                       host_regex_table[0]=0;   /*in case of one regex offset must be 0*/
    190 +                       cudaMalloc((void**)&device_regex_table, sizeof (u32) );
    191 +                       cudaMemcpy(device_regex_table, host_regex_table, sizeof(u32), cudaMemcpyHostToDevice);/
    192 +                       pMatch(device_line, device_line_table, num_lines, 1, timerOn, device_regex, device_regex
    193 +
    194 +
185 195                         endPMatch = CycleTimer::currentSeconds();
186 196                 }
187 197                 // match a bunch of regexs
```

```
13      -__device__ State *states;
14      -__device__ static int pnstate;
    13  +__device__ __shared__ State *states; /*this variable used in ppost2nfa
    14  +                          it must be local for each block
    15  +                          simple way to do this use __shared__
    16  +                        */
    17  +__device__ __shared__  int pnstate; /*this variable too must be local
    18  +                                     for each blocks*/
    19  +
15  20  __device__ State pmatchstate = { Match };      /* matching state */
16  21
17  22
```

Замечания о CW,DFA,RK

- Commentz-Walter
  - Самый медленный из этой тройки.
  - Trie для CW строится быстрее всего.
- DFA
  - быстрее CW, но медленнее Rabin-Karp
  - автомат строится медленнее, чем Trie для CW, однако быстрее ищет.
  - На более чем 1100-500 словах спад производительности с 1-2гигабит/сек до 2-60 мегабит/сек.
- Rabin-Karp
  - Самый быстрый из трех рассмотренных
  - Поиск хэш функции занимает больше времени, чем построение Trie и конечного автомата.

Особенности нашего подхода

В этой работе будет представлен алгоритм Rabin-Karp
с применением техники FFDM(будет описано далее)
и хэш-функцией hash404(которую придумали авторы доклада).

```
subsP={P_1..P_n}
hashes={h(P_1),...,h(P_n)}
RK_search(T[0..N], subsP, hashes) {
  pre_search(T[0..l_s-2],subsP)
  for(j=l_s-1;j<=N;j++){
    h_1 = h( T[j-l_1+1,..,j] )
      ...
    h_s = h( T[j-l_s+1,..,j] )

    if (h_1 ∈ hashes)
      if (T[j-l_1+1,..,j] ∈ subsP)
        print j, T[j-l_1+1,..,j]
          ...
    if (h_s ∈ hashes)
      if (T[j-l_s+1,..,j] ∈ subsP)
        print j, T[j-l_s+1,..,j]
  }
}
```

Для перехода от общей идеи к конкретной, нужно внести определенность в три вопроса:

- Какой формулой задается хэш-функция `h()`?
- Что представляет из себя операция $\in$ для множества `hashes`.
- Что представляет из себя операция $\in$ для множества `subsP`.

Описание предлагаемого алгоритма, хэш-функция

Стоит обратить внимание на 2 факта.

1. Хэш функция пересчитывается каждый раз при переходе к следующей букве текста.
2. Если среди шаблонов встречаются s разных длин, тогда хэш будет вычисляться $\sim |T| \times s$ раз.

Из этого вытекает, что больш*ая* часть вычислений будет приходиться на вычисление хэш-функции.

В связи с этим от хэш-функции нужно потребовать, чтобы она вычислялась на основе своего значения с предыдущего шага и новой буквы.

h_k^j=h(T[j-l_k+1..j]) $\longleftarrow$ Медленно!

h_k^j=h(T[j],h_k^{j-1}) $\longleftarrow$ Быстро!

Если хэш вычисляется на основе предыдущего значения и новой буквы, то его называют rolling hash.

Известные варианты rolling hash.

Поскольку всякая буква имеет свое бинарное представление, будем рассматривать буквы как целые числа.

- Rabin-Karp rolling hash
  $h(c_1..c_k) = c_1 a^{k-1} + ... + c_{k-1} a + c_k$
  $h(c_2..c_{k+1}) = (h(c_1...c_k) - c_1 a^{k-1})a + c_{k+1}$

- Cyclic polynomial rolling hash
  $h(c_1..c_k) = f(c_1)\texttt{<<}(k-1) \hat{} .. \hat{} f(c_{k-1})\texttt{<<}1 \hat{} f(c_k)$
  $h(c_2..c_{k+1}) = \Big(\big(h(c_1..c_k) \hat{} f(c_1)\texttt{<<}(k-1)\big)\texttt{<<}1\Big) \hat{} f(c_{k+1})$
  где $f(c_i)$ — некоторая функция, которая обычно использует память.

Описание предлагаемого алгоритма, хэш-функция

Предлагается взять следующую функцию на роль функции h().

$$\texttt{h}(\texttt{a}_1 \ldots \texttt{a}_k)=((\ (\ldots((\texttt{a}_1<<\texttt{m}_1)\hat{}\texttt{a}_2)\ldots)<<\texttt{m}_{k-1})\hat{}\texttt{a}_k\ )\ \&\ \texttt{mask} =$$
$$\left(\texttt{a}_1<<\left(\sum_{i=1}^{k-1} m_i\right)\right)\hat{}\left(\texttt{a}_2<<\left(\sum_{i=2}^{k-1} m_i\right)\right)\hat{}\ldots\hat{}\left(\texttt{a}_{k-1}<<\left(\sum_{i=k-1}^{k-1} m_i\right)\right)\hat{}\texttt{a}_k\ \&\ \texttt{mask}$$

В записи функции h() были использованы следующие обозначения.

- $<<\texttt{m}_k$ — битоый сдвиг влево на $\texttt{m}_k$ позиций
- ^ — XOR (побитовое сложение по модулю 2)
- & — Битовое И
- mask имеет вид 2^M-1

Пример. $\texttt{h}(\texttt{a}_1\texttt{a}_2\texttt{a}_3) = ((((\texttt{a}_1<<\texttt{m}_1)\hat{}\texttt{a}_2)<<\texttt{m}_2)\hat{}\texttt{a}_3)\ \&\ \texttt{mask}$

Чтобы пересчитывать этот хэш на основе предыдущего предлагается следующая схема вычислений:

Описание предлагаемого алгоритма, хэш-функция

Сначала будет представлен алгоритм вычисления хэша, а затем формальное обоснование корректности.

1. Вводится массив h длины l_s. Будем рассматривать элементы этого массива в моменты времени 0,1,2,..

   Элементы массива будут обозначаться h_i^j, здесь j — время(номер шага), i — порядковый номер элемента в массиве.

2. Перед началом поиска, на нулевом шаге, массив можно оставить неинициализированным:

   $$h_i^0 = *,\ i \in [1..l_s]$$

3. Далее пересчет осуществляется по формуле

   $$h_i^j = (h_{i-1}^{j-1} << m_{i-1}) \hat{}\ a_j\ \&\ mask,\ i \in [l_s..2]$$

   Вычисления идут от старшего индекса l_s к 2.

4. И затем первый элемент массива опередеяется следующим соотношением

   $$h_1^j = a_j\ \&\ mask$$

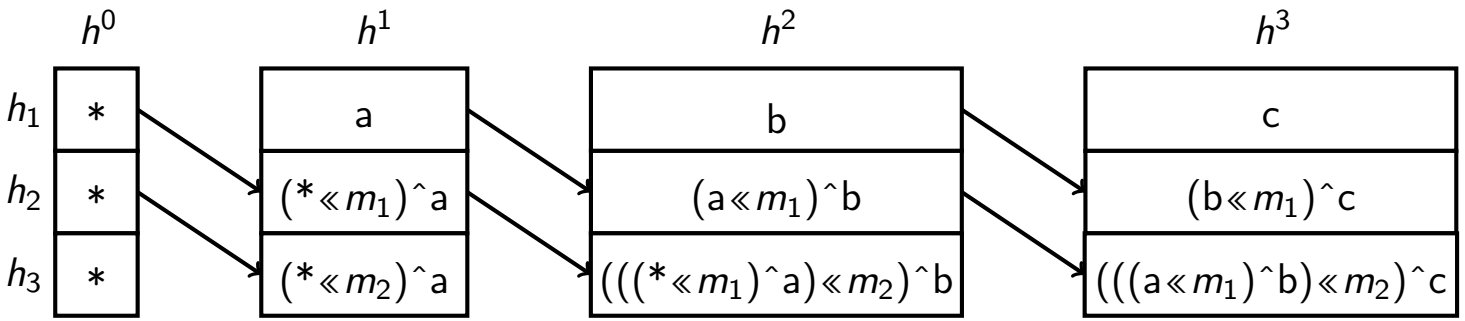
5. после прохода по l_s символам в ячейках массива h на местах $l_1, l_2, .., l_s$ будут храниться искомые хэши.

Описание предлагаемого алгоритма, hash404

Оба представления предлагаемой хэш-функция далее будет называться `hash404()`

1. $\texttt{hash404}_*(a_1..a_s) = (((..((a_1 << m_1) \hat{} a_2)..) << m_{s-1}) \hat{} a_s)$ `&` $\texttt{mask}$
2. `hash404(h,a`$_j$`):`
   $h_s^0 = *;$
   $h_n^j = ((h_{n-1}^{j-1} << m_{n-1}) \hat{} a_j)$ `&` $\texttt{mask};$
   $h_1^j = a_j$ `&` $\texttt{mask};$

Пример вычисления hash404 от строки "abc".



К каждой ячейке массива нужно применять маску.

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \leq q \leq N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений v_1=hash404$_*$($a_1..a_q$), v_2=h[q], где h получается в результате последовательности действий h^1=hash404(h^0,a_1)$\mapsto$...$\mapsto h^q$=hash404(h^{q-1},a_q).
Тогда v_1=v_2. То есть в h[q] хранится значение hash404$_*$($a_1..a_q$).

Замечание. Если применить сформулированное утверждение для букв $a_{s+1}..a_q$, то получится, что h[q-s] будет хранить значение hash404$_*$($a_{s+1}..a_q$).
Замечание. Если при помощи hash404 и начального вектора h^0 обработано q букв, то нельзя использовать значение h[q+s], $s \geq 1$, поскольку в них будет информация о h^0.

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \leq q \leq N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений $v_1=\text{hash404}_*(a_1..a_q)$, $v_2=h[q]$, где h получается в результате последовательности действий
$h^1=\text{hash404}(h^0,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Тогда $v_1=v_2$. То есть в h[q] хранится значение $\text{hash404}_*(a_1..a_q)$.

Доказательство. Пусть h^q — экземпляр массива h, после применения операторов $h^1=\text{hash404}(h,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Докажем утверждение индукцией по q.
База. q=1.
$\text{hash404}_*(a_1)=a_1$ `& mask`.
$h^1=\text{hash404}(h,a_1)$, $h^1[1]=a_j$ `& mask`

$\Downarrow$

$\text{hash404}_*(a_1)=h^1[1]$.
База доказана.

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \leq q \leq N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений $v_1=\text{hash404}_*(a_1..a_q)$, $v_2=h[q]$, где h получается в результате последовательности действий $h^1=\text{hash404}(h^0,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Тогда $v_1=v_2$. То есть в h[q] хранится значение $\text{hash404}_*(a_1..a_q)$.

Доказательство. Пусть h^q — экземпляр массива h, после применения операторов $h^1=\text{hash404}(h,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Докажем утверждение индукцией по q.
Пусть для q-1 верно утверждение.

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \le q \le N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений $v_1 = \text{hash404}_*(a_1..a_q)$, $v_2 = h[q]$, где h получается в результате последовательности действий
$h^1 = \text{hash404}(h^0, a_1) \mapsto ... \mapsto h^q = \text{hash404}(h^{q-1}, a_q)$.
Тогда $v_1 = v_2$. То есть в h[q] хранится значение $\text{hash404}_*(a_1..a_q)$.

Доказательство. Пусть h^q — экземпляр массива h, после применения операторов $h^1 = \text{hash404}(h, a_1) \mapsto ... \mapsto h^q = \text{hash404}(h^{q-1}, a_q)$.
Докажем утверждение индукцией по q.
Докажем, что верно и при q.
h^{q-1}[q-1] = $\text{hash404}_*(a_1..a_{q-1})$
$h^q[q] = (h^{q-1}[q\text{-}1] \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$
$(\text{hash404}_*(a_1..a_{q-1}) \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$

$$\Big((\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1}\ ) \ \&\ \text{mask} \Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$$\Big((\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1}\ ) \ \Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$\text{hash404}_*(a_1..a_q)$

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \le q \le N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений $v_1=\text{hash404}_*(a_1..a_q)$, $v_2=h[q]$, где h получается в результате последовательности действий $h^1=\text{hash404}(h^0,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Тогда $v_1=v_2$. То есть в h[q] хранится значение $\text{hash404}_*(a_1..a_q)$.

Доказательство. Пусть h^q — экземпляр массива h, после применения операторов $h^1=\text{hash404}(h,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Докажем утверждение индукцией по q.
Докажем, что верно и при q.
$h^{q-1}[q\text{-}1] = \text{hash404}_*(a_1..a_{q-1})$ — по предположению индукции имеем
$h^q[q] = (h^{q-1}[q\text{-}1] \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$
$(\text{hash404}_*(a_1..a_{q-1}) \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$
$\Big(((...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1}) \ \&\ \text{mask}\Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$
$\Big(((...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1})\Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$
$\text{hash404}_*(a_1..a_q)$

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \le q \le N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений $v_1=\text{hash404}_*(a_1..a_q)$, $v_2=h[q]$, где h получается в результате последовательности действий $h^1=\text{hash404}(h^0,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Тогда $v_1=v_2$. То есть в h[q] хранится значение $\text{hash404}_*(a_1..a_q)$.

Доказательство. Пусть h^q — экземпляр массива h, после применения операторов $h^1=\text{hash404}(h,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Докажем утверждение индукцией по q.
Докажем, что верно и при q.
$h^{q-1}[q\text{-}1] = \text{hash404}_*(a_1..a_{q-1})$
$h^q[q] = (h^{q-1}[q\text{-}1] \ll m_{q-1})\hat{}\,a_q \;\&\; \text{mask} =$
$(\text{hash404}_*(a_1..a_{q-1}) \ll m_{q-1})\hat{}\,a_q \;\&\; \text{mask} =$

по определению hash404 имеем тождество

$$\Big(\big(\ (...((a_1 \ll m_1)\hat{}\,a_2)...) \ll m_{q-2}\big)\hat{}\,a_{q-1}\ \big) \;\&\; \text{mask}\Big) \ll m_{q-1}\hat{}\,a_q \;\&\; \text{mask} =$$

$$\Big(\big(\ (...((a_1 \ll m_1)\hat{}\,a_2)...) \ll m_{q-2}\big)\hat{}\,a_{q-1}\ \big)\Big) \ll m_{q-1}\hat{}\,a_q \;\&\; \text{mask} =$$

$\text{hash404}_*(a_1..a_q)$

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \le q \le N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений $v_1=\text{hash404}_*(a_1..a_q)$, $v_2=h[q]$, где h получается в результате последовательности действий $h^1=\text{hash404}(h^0,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Тогда $v_1=v_2$. То есть в h[q] хранится значение $\text{hash404}_*(a_1..a_q)$.

Доказательство. Пусть h^q — экземпляр массива h, после применения операторов $h^1=\text{hash404}(h,a_1)\mapsto...\mapsto h^q=\text{hash404}(h^{q-1},a_q)$.
Докажем утверждение индукцией по q.
Докажем, что верно и при q.
$h^{q-1}[q\text{-}1] = \text{hash404}_*(a_1..a_{q-1})$
$h^q[q] = (h^{q-1}[q\text{-}1] \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$
$(\text{hash404}_*(a_1..a_{q-1}) \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$

подставляем первое равенство во второе

$$\Big(\big(\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}\big) \hat{} a_{q-1}\ \Big) \ \&\ \text{mask}\Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$$\Big(\big(\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}\big) \hat{} a_{q-1}\ \Big)\Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$\text{hash404}_*(a_1..a_q)$

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \le q \le N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений $v_1 = \text{hash404}_*(a_1..a_q)$, $v_2 = h[q]$, где h получается в результате последовательности действий
$h^1 = \text{hash404}(h^0, a_1) \mapsto ... \mapsto h^q = \text{hash404}(h^{q-1}, a_q)$.
Тогда $v_1 = v_2$. То есть в h[q] хранится значение $\text{hash404}_*(a_1..a_q)$.

Доказательство. Пусть h^q — экземпляр массива h, после применения операторов $h^1 = \text{hash404}(h, a_1) \mapsto ... \mapsto h^q = \text{hash404}(h^{q-1}, a_q)$.
Докажем утверждение индукцией по q.
Докажем, что верно и при q.
$h^{q-1}[q\text{-}1] = \text{hash404}_*(a_1..a_{q-1})$
$h^q[q] = (h^{q-1}[q\text{-}1] \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$
$(\text{hash404}_*(a_1..a_{q-1}) \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$

раскрываем hash404_* по определению

$$\Big((\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1}\ ) \ \&\ \text{mask} \Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$$\Big((\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1}\ ) \Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$\text{hash404}_*(a_1..a_q)$

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \le q \le N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений $v_1 = \text{hash404}_*(a_1..a_q)$, $v_2 = h[q]$, где h получается в результате последовательности действий
$h^1 = \text{hash404}(h^0, a_1) \mapsto ... \mapsto h^q = \text{hash404}(h^{q-1}, a_q)$.
Тогда $v_1 = v_2$. То есть в h[q] хранится значение $\text{hash404}_*(a_1..a_q)$.

Доказательство. Пусть h^q — экземпляр массива h, после применения операторов $h^1 = \text{hash404}(h, a_1) \mapsto ... \mapsto h^q = \text{hash404}(h^{q-1}, a_q)$.
Докажем утверждение индукцией по q.
Докажем, что верно и при q.
h^{q-1}[q-1] = $\text{hash404}_*(a_1..a_{q-1})$
$h^q[q] = (h^{q-1}[q\text{-}1] \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$
$(\text{hash404}_*(a_1..a_{q-1}) \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$

Убирем mask внутри больших скобок
$\text{mask} = 2^M - 1$
$((A \& \text{mask}) \ll k) \& \text{mask} =$
$((A \% 2^M) 2^k) \% 2^M = (A 2^k) \% 2^M =$
$(A \ll k) \ \&\ \text{mask}$

$$\Big((\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1}\ ) \ \&\ \text{mask} \Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$$\Big((\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1}\ ) \Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$\text{hash404}_*(a_1..a_q)$

Описание предлагаемого алгоритма, hash404

Утверждение . Пусть массив h имеет размер N, $1 \le q \le N$ и $a_1..a_q$ — любые, h^0 — произвольный вектор. Пусть числа v_1, v_2 получаются из соотношений $v_1 = \text{hash404}_*(a_1..a_q)$, $v_2 = h[q]$, где h получается в результате последовательности действий $h^1 = \text{hash404}(h^0, a_1) \mapsto ... \mapsto h^q = \text{hash404}(h^{q-1}, a_q)$.
Тогда $v_1 = v_2$. То есть в h[q] хранится значение $\text{hash404}_*(a_1..a_q)$.

Доказательство. Пусть h^q — экземпляр массива h, после применения операторов $h^1 = \text{hash404}(h, a_1) \mapsto ... \mapsto h^q = \text{hash404}(h^{q-1}, a_q)$.
Докажем утверждение индукцией по q.
Докажем, что верно и при q.
$h^{q-1}[q\text{-}1] = \text{hash404}_*(a_1..a_{q-1})$
$h^q[q] = (h^{q-1}[q\text{-}1] \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$
$(\text{hash404}_*(a_1..a_{q-1}) \ll m_{q-1}) \hat{} a_q \ \&\ \text{mask} =$

Сворачиваем выражение по определению hash404_*

$$\Big((\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1}\ ) \ \&\ \text{mask} \Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$$\Big((\ (...((a_1 \ll m_1) \hat{} a_2)...) \ll m_{q-2}) \hat{} a_{q-1}\ ) \ \Big) \ll m_{q-1} \hat{} a_q \ \&\ \text{mask} =$$

$\text{hash404}_*(a_1..a_q)$

Описание предлагаемого алгоритма, хэш-функция, C

6 XOR + 6 SHIFTS + 1 AND на каждый символ текста

```
define hash404(h,w) do{ \
 h[6]=( (h[5] << shifts[6]) ^ (unsigned)( (unsigned char)(*w)))&mask;\
 h[5]=( (h[4] << shifts[5]) ^ (unsigned)( (unsigned char)(*w)) );\
 h[4]=( (h[3] << shifts[4]) ^ (unsigned)( (unsigned char)(*w)) );\
 h[3]=( (h[2] << shifts[3]) ^ (unsigned)( (unsigned char)(*w)) );\
 h[2]=( (h[1] << shifts[2]) ^ (unsigned)( (unsigned char)(*w)) );\
 h[1]=( (h[0] << shifts[1]) ^ (unsigned)( (unsigned char)(*w)) );\
 h[0]=(                       (unsigned)( (unsigned char)(*w)) );\
}while(0)
```

Сравнение хэш-функций.

- $\texttt{rabinkarp}(c_1..c_k) = c_1a^{k-1} + ... + c_{k-1}a + c_k$
- $\texttt{cpoly}(c_1..c_k)=\texttt{f}(c_1)\texttt{<<(k-1) ^..^ f}(c_{k-1})\texttt{<<1 ^ f}(c_k)$
- $\texttt{hash404}_*(c_1..c_k)=(((..((c_1\texttt{<<}m_1)\texttt{^}c_2)..)\texttt{<<}m_{k-1})\texttt{^}c_k)$

1. Производительность хэш-функции(Megabit/sec).
2. На скольки словах будет первая коллизия.

Описание предлагаемого алгоритма, pref+suff

Замечание. Шаблоны имеют длины $l_1 < ... < l_s$. Если число s велико, тогда на каждый символ текста придется делать много проверок типа (*) (**). Это пагубно влияет на производительность.

(*) `if (`$h_{l_i} \in$ `hashes)`

(**) `if (T[j-`l_i`+1,..,j]` $\in$ `subsP)`

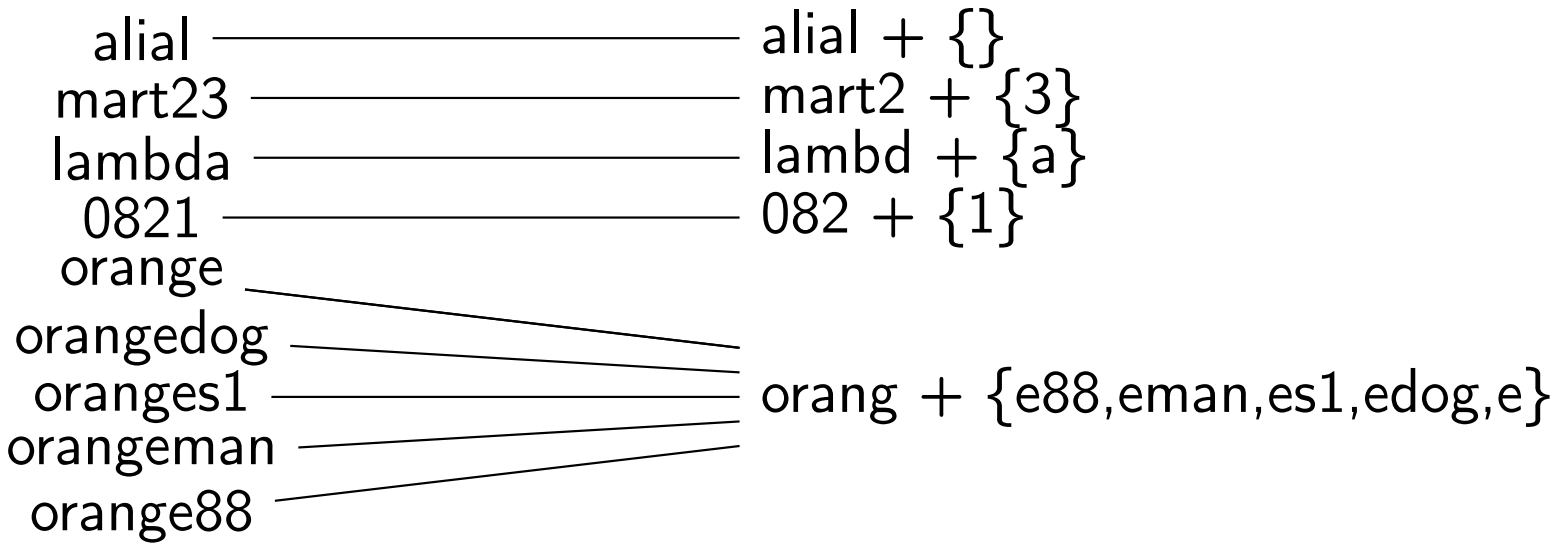
`print j, T[j-`l_i`+1,..,j]`

Если число s велико, то предлагается взять $p < s$ и новые длины $d_1, .., d_p$. После чего от каждого шаблона брать префикс длины d_i. И вместо шаблонов искать префиксы. В случае совпадения префикса сохранять позицию. А после прохода по некоторому участку текста проверять суффиксы.

Замечание. Далее будет показано, как сильно будет влиять на производительность выбор p и $\{d_i\}$

Описание предлагаемого алгоритма, pref+suff

- В левой колонки расположены шаблоны, от которых будут выделяться префиксы длин 3 и 5.
- В правой колонке префикс + множество суффиксов.
- От шаблона будет отделяться префикс максимально возможной длины.



	hash404	cyclicpoly	rk(a=2^n+1)	rk(a=2^n)
3	11628.686266	8975.666834	5459.508784	9075.816427
3,4	8155.304471	6701.642626	5286.852478	6807.731877
3,5	6773.024824	5237.075410	5434.481972	6809.373420
3,6	5451.320818	5406.709152	4873.773626	9064.462412
3,7	4955.929062	6505.687006	5165.668785	9069.239586
3,8	4548.478312	6503.727693	5165.135807	9082.477714
3,9	3902.020016	6492.991673	5166.759309	9056.859186
3,10	3409.894136	6502.230191	5163.900690	9078.060712
3,4,5	6780.739355	5430.837439	4195.745543	4961.913000
3,4,6	5451.509702	4188.501154	3899.229415	6807.058652
3,4,7	4960.303956	4872.221228	4181.265865	6806.049065
3,4,8	4551.222571	4872.005696	4168.952057	6805.334122
3,4,9	3901.646784	4879.149969	4163.735114	6822.576583
3,4,10	3411.964657	4874.054025	4145.398744	6819.788272
3,5,6	5443.184277	3913.514489	3898.980941	6054.053186
3,5,7	4955.951363	4162.318867	3901.024889	6060.082688
3,5,8	4549.267417	4150.475961	3898.815310	6056.883226
3,5,9	3897.711459	4154.734770	3896.415233	6056.150491
3,5,10	3414.270538	4142.950505	3896.580660	6058.549191
3,6,7	4959.946532	3873.761018	3898.594490	8748.441080

	hash404	rk($a = 2^n$)
3	15454	15454
3,4	15454	15454
3,5	15454	3510
3,6	13889	11674
3,7	12823	534
3,8	8077	534
3,9	15454	1127
3,10	15454	2242
3,4,5	15454	3510
3,4,6	13889	11674
3,4,7	12772	893
3,4,8	8077	534
3,4,9	15454	1714
3,4,10	15454	2331
3,5,6	13889	11674
3,5,7	12772	3471
3,5,8	6903	544
3,5,9	10750	1714
3,5,10	12940	2331
3,6,7	15454	3471

- В таблице количество тестовых шаблонов, которых удалось отобразить хэш-функцией без коллизий.
- rk(a=2^n) – RabinKarp rolling hash, a=2^n. Перебирались все $n \in [0..31]$.
- У hash404 перебирались все возможные сдвиги $\{m_i\}$

Описание предлагаемого алгоритма, операция $\in$

Нужно определить операцию $\in$ для множества hashes и subsP.

(*) `if (`h_{l_i} `∈ hashes)`

(**) `if (T[j-`l_i`+1,..,j] ∈ subsP)`

`print j, T[j-`l_i`+1,..,j]`

Самое простое решение. Завести пару массивов, длиной как максимальное значение хэш-функции $h_{\max}$

- for(...) hashes[i]=NIL;
- hashes[hash(P_i)]=hash(P_i)
- Тогда $\forall$ h $\in$ *Im*(*hash*) верно h $\in$ hashes $\Leftrightarrow$ h==hashes[h]

Однако памяти не хватит.

Для проверки (**) тоже можно поступить аналогично и аналогично памяти не хватит.

Операция $\in$ и First Fit Decrease Method

FFDM применяется для решения следующей задачи.
Пусть есть пары (key_i,val_i), $i \in [1..n]$
key_i — попарно различные числа.
val_i — произвольные объекты.
Требуется реализовать функцию

$$\text{ffdm} : [0..N] \to \{\text{val}_1, .., \text{val}_n\}$$

$$\text{ffdm}(k) = \begin{cases} \text{val}_i & , \quad \text{if } k == \text{key}_i \\ \text{произв}, & \quad \text{иначе} \end{cases}$$

Эту технику можно использовать для проверок (*), (**).

$\texttt{hashes=\{hash(P}_1\texttt{),...,hash(P}_n\texttt{)\}}$

Пусть hash инъективен на subsP.

(*) $\texttt{key}_i\texttt{=val}_i\texttt{=hash(P}_i\texttt{)}$

$\texttt{h} \in \texttt{hashes} \Leftrightarrow \texttt{h==ffdm(h)}$

(**) $\texttt{key}_i\texttt{=hash(P}_i\texttt{)}\ \texttt{val}_i\texttt{=P}_i$

$\texttt{S} \in \texttt{subsP} \Leftrightarrow \texttt{S==ffdm(hash(S))}$

Утверждение 1.
Пусть $f : A \to B, B \subset A$,
$Im(f) = B, \forall b \in B\ f(b) = b$
Тогда $h \in B \Leftrightarrow f(h) = h$

$$\text{ffdm} : [0..N] \to \{\text{val}_1, .., \text{val}_n\}$$

$$\text{ffdm}(k) = \begin{cases} \text{val}_i, & \text{if } k == \text{key}_i \\ \text{произв}, & \text{иначе} \end{cases}$$

Доказательство.

1. Пусть f(h)=h $\Rightarrow$ B $\ni$ f(h) = h
2. Пусть h $\in$ B $\Rightarrow$ f(h) = h по условию.

Утверждение 2. Пусть key_i=hash(P_i), val_i=hash(P_i),
тогда `h` $\in$ `hashes` $\Leftrightarrow$ `h==ffdm(h)`
Доказательство. A = $\mathbb{Z}$, B=hashes, $B \subset A$, Im(ffdm)=hashes
$\forall h \in B$, ffdm(h)=ffdm(hash(P_j))=hash(P_j)=h. Берем f=ffdm.
По утв. 1 получаем требуемое.
Утверждение 3. Пусть key_i=hash(P_i), val_i=P_i,
тогда S $\in$ subsP $\Leftrightarrow$ S==ffdm(hash(S))
Доказательство. $A = \Sigma^*, B = \text{subsP}, f = \text{ffdm} \circ \text{hash}$,
$\text{Im}(f) = \text{subsP} = B$
$\forall P_j = b \in B\ f(b) = f(P_j) = \text{ffdm}(\text{hash}(P_j)) = P_j = b$
По утв. 1 получаем требуемое.

Построение отображения ffdm

1. Выбираем число t, удовлетворяющее условию $key_{max} < t^2$
2. Рассматриваем квадратную матрицу размера t×t.
3. Помещаем каждое val_i в эту матрицу. Позиция определяется числами (x,y), где $y=key_i/t$, и $x=key_i \bmod t$
4. Будем сдвигать очередную строку вправо, пока никакой ее элемент не будет лежать в одном столбце с элементами из предыдущих строк. Смещения кладем в массив r.
5. После таких сдвигов в каждом столбце матрицы будет не более одного элемента. Проецируем матрицу на линейный массив C: столбец→элемент массива.

Теперь для вычисления функции нужно проделать следующие действия.

y=key/t
x=key mod t
index=r[y]+x
ffdm(key)=C[index]

FFDM,пример

K={0, 3, 4, 7, 10, 13, 15, 18, 19, 21, 22, 24, 26, 29, 30, 34}
V={v_0,v_3,v_4,v_7,v_{10},v_{13},v_{15},v_{18},v_{19},v_{21},v_{22},v_{24},v_{26},v_{29},v_{30},v_{34}}

```
A |    0    1    2    3     4    5
--+-------------------------------
0 |    .    .    .    .     .    .
1 |    .    .    .    .     .    .
2 |    .    .    .    .     .    .
3 |    .    .    .    .     .    .
4 |    .    .    .    .     .    .
5 |    .    .    .    .     .    .
```

1. Выбираем число t, удовлетворяющее условию $key_{max} < t^2$
$key_{max} = 34$
Берем t=6

```
r[0]=0   v0  .  .   v3  v4   .
r[1]=1       .  v7  .   .   v10   .
r[2]=5                  .   v13  .  v15   .    .
r[3]=9                                  v18  v19   .   v21  v22   .
r[4]=14                                                          v24  .  v26  .  .  v29
r[5]=7                             v30  .    .    .   v34   .

vals:  v0 . v7 v3 v4 v10 v13 v30 v15 v18 v19 v34 v21 v22 v24 .  v26 .  .  v29
index: 0  1 2  3  4  5   6   7   8   9   10  11  12  13  14  15 16  17 18 19
```

FFDM,пример

K={0, 3, 4, 7, 10, 13, 15, 18, 19, 21, 22, 24, 26, 29, 30, 34}
V={v_0,v_3,v_4,v_7,v_{10},v_{13},v_{15},v_{18},v_{19},v_{21},v_{22},v_{24},v_{26},v_{29},v_{30},v_{34}}

A	0	1	2	3	4	5
0	.	.	.	.	.	.
1	.	.	.	.	.	.
2	.	.	.	.	.	.
3	.	.	.	.	.	.
4	.	.	.	.	.	.
5	.	.	.	.	.	.

2. Рассматриваем квадратную матрицу, размера t×t., t=6

```
r[0]=0   v0  .  .   v3  v4   .
r[1]=1       .  v7  .   .   v10   .
r[2]=5                  .   v13  .  v15   .    .
r[3]=9                                  v18  v19   .   v21  v22   .
r[4]=14                                                          v24  .  v26  .  .  v29
r[5]=7                             v30  .    .    .   v34   .

vals:  v0 . v7 v3 v4 v10 v13 v30 v15 v18 v19 v34 v21 v22 v24 .  v26 .  .  v29
index: 0  1 2  3  4  5   6   7   8   9   10  11  12  13  14  15 16  17 18 19
```

FFDM,пример

K={0, 3, 4, 7, 10, 13, 15, 18, 19, 21, 22, 24, 26, 29, 30, 34}
V={v_0,v_3,v_4,v_7,v_{10},v_{13},v_{15},v_{18},v_{19},v_{21},v_{22},v_{24},v_{26},v_{29},v_{30},v_{34}}

A	0	1	2	3	4	5
0	v0	.	.	v3	v4	.
1	.	v7	.	.	v10	.
2	.	v13	.	v15	.	.
3	v18	v19	.	v21	v22	.
4	v24	.	v26	.	.	v29
5	v30	.	.	.	v34	.

3. Помещаем каждое val_i в эту матрицу. Позиция определяется числами (x,y), где y=key_i/t, и x=key_i mod t

```
r[0]=0   v0 .  .   v3 v4   .
r[1]=1       . v7 .   .   v10   .
r[2]=5                    .   v13 . v15   .    .
r[3]=9                                  v18 v19   .   v21 v22   .
r[4]=14                                                       v24 . v26 . . v29
r[5]=7                             v30 .    .    .   v34   .

vals:  v0 . v7 v3 v4 v10 v13 v30 v15 v18 v19 v34 v21 v22 v24 .  v26 .  .  v29
index: 0  1 2  3  4  5   6   7   8   9   10  11  12  13  14  15 16  17 18 19
```

FFDM,пример

K={0, 3, 4, 7, 10, 13, 15, 18, 19, 21, 22, 24, 26, 29, 30, 34}
V={v_0,v_3,v_4,v_7,v_{10},v_{13},v_{15},v_{18},v_{19},v_{21},v_{22},v_{24},v_{26},v_{29},v_{30},v_{34}}

```
A |    0    1    2    3     4    5
--+-------------------------------
0 |   v0    .    .   v3    v4    .
1 |    .   v7    .    .    v10   .
2 |    .   v13   .   v15    .    .
3 |   v18  v19   .   v21   v22   .
4 |   v24   .   v26   .     .   v29
5 |   v30   .    .    .    v34   .
```

4. Будем сдвигать очередную строку вправо, пока никакой ее элемент не будет лежать в одном столбце с элементами из предыдущих строк.

```
r[0]=0   v0 .  .   v3 v4   .
r[1]=1       .  v7 .  .   v10  .
r[2]=5                .   v13 .  v15  .    .
r[3]=9                                 v18 v19  .   v21 v22  .
r[4]=14                                                      v24 .  v26 .  .  v29
r[5]=7                                  v30 .    .    .   v34  .
```

```
vals:  v0 . v7 v3 v4 v10 v13 v30 v15 v18 v19 v34 v21 v22 v24 .  v26 .  .  v29
index: 0  1 2  3  4  5   6   7   8   9   10  11  12  13  14  15 16  17 18 19
```

FFDM,пример

K={0, 3, 4, 7, 10, 13, 15, 18, 19, 21, 22, 24, 26, 29, 30, 34}
V={v_0,v_3,v_4,v_7,v_{10},v_{13},v_{15},v_{18},v_{19},v_{21},v_{22},v_{24},v_{26},v_{29},v_{30},v_{34}}

```
A |    0    1    2    3     4    5
--+-------------------------------
0 |   v0    .    .   v3    v4    .
1 |    .   v7    .    .    v10   .
2 |    .   v13   .   v15    .    .
3 |   v18  v19   .   v21   v22   .
4 |   v24   .   v26   .     .   v29
5 |   v30   .    .    .    v34   .
```

5. После таких сдвигов в каждом столбце матрицы будет не более одного элемента. Проецируем матрицу на линейный массив: столбец→элемент массива.

```
r[0]=0   v0 .  .   v3 v4   .
r[1]=1       . v7 .   .   v10   .
r[2]=5                    .   v13 .  v15   .    .
r[3]=9                                  v18 v19   .   v21 v22   .
r[4]=14                                                       v24 .  v26 .  .  v29
r[5]=7                             v30 .    .    .   v34   .
```

```
vals:  v0 . v7 v3 v4 v10 v13 v30 v15 v18 v19 v34 v21 v22 v24 .  v26 .  .  v29
index: 0  1 2  3  4  5   6   7   8   9   10  11  12  13  14  15 16  17 18 19
```

FFDM,пример

K={0, 3, 4, 7, 10, 13, 15, 18, 19, 21, 22, 24, 26, 29, 30, 34}
V={v_0,v_3,v_4,v_7,v_{10},v_{13},v_{15},v_{18},v_{19},v_{21},v_{22},v_{24},v_{26},v_{29},v_{30},v_{34}}

```
A |    0    1    2    3     4    5
--+-------------------------------
0 |   v0    .    .   v3    v4    .
1 |    .   v7    .    .    v10   .
2 |    .   v13   .   v15    .    .
3 |   v18  v19   .   v21   v22   .
4 |   v24   .   v26   .     .   v29
5 |   v30   .    .    .    v34   .
```

6. Что бы получить val_i по key_i нужно использовать вот эту формулу:
y=key/t
x=key mod t
index=r[y]+x
ffdm(key)=C[index]

```
r[0]=0   v0 .  .   v3 v4   .
r[1]=1       .  v7 .   .   v10   .
r[2]=5                     .   v13 .  v15   .    .
r[3]=9                                    v18 v19   .   v21 v22   .
r[4]=14                                                           v24 .  v26 .  .  v29
r[5]=7                                v30 .    .    .   v34   .
```

```
vals:  v0 . v7 v3 v4 v10 v13 v30 v15 v18 v19 v34 v21 v22 v24 .  v26 .  .  v29
index: 0  1 2  3  4  5   6   7   8   9   10  11  12  13  14  15 16  17 18 19
```

Релизация вычислительного ядра

```
for(unsigned i=0;i<textlen-6;i++) {
 hash404(h,w);
 if( ffdm_checkhash(h[2]) ) {
   unsigned offset3=ffdm_get_offset(h[2]);
   if(/*проверяем совпадение букв*/
      (w[ 0] == (words+offset3)[2]) &&
      (w[-1] == (words+offset3)[1]) &&
      (w[-2] == (words+offset3)[0]) &&
               !(words+offset3)[3]
    )
   {/* нашлось вхождение. */}
 }
 if( ffdm_checkhash(h[6]) ) {
   unsigned offset7=ffdm_get_offset(h[6]);
   if(
      (w[ 0] == (words+offset7)[6]) &&
      (w[-1] == (words+offset7)[5]) &&
      (w[-2] == (words+offset7)[4]) &&
      (w[-3] == (words+offset7)[3]) &&
      (w[-4] == (words+offset7)[2]) &&
      (w[-5] == (words+offset7)[1]) &&
      (w[-6] == (words+offset7)[0]) &&
               !(words+offset7)[7]
     )
    {/* нашлось вхождение. */}
  }
  w++;
}
```

На этом слайде представлено вычислительное ядро, которое ищет префиксы от шаблонов.

(*) В рамках этой работы был написан генератор, который генерирует такие вычислительные ядра.
(*) Префиксы имеют длины 3 и 7.
(*) Перед этим циклом был обработан начальный кусок текста длиной 6 на предмет вхождения шаблонов длины 3.
(*) цикл начинает работу с 7 символа и проходит по всему тексту.
(*) w - указатель на текст.

Релизация вычислительного ядра

```
for(unsigned i=0;i<textlen-6;i++) {
 hash404(h,w);
 if( ffdm_checkhash(h[2]) ) {
   unsigned offset3=ffdm_get_offset(h[2]);
   if(/*проверяем совпадение букв*/
      (w[ 0] == (words+offset3)[2]) &&
      (w[-1] == (words+offset3)[1]) &&
      (w[-2] == (words+offset3)[0]) &&
               !(words+offset3)[3]
    )
   {/* нашлось вхождение. */}
 }
 if( ffdm_checkhash(h[6]) ) {
   unsigned offset7=ffdm_get_offset(h[6]);
   if(
      (w[ 0] == (words+offset7)[6]) &&
      (w[-1] == (words+offset7)[5]) &&
      (w[-2] == (words+offset7)[4]) &&
      (w[-3] == (words+offset7)[3]) &&
      (w[-4] == (words+offset7)[2]) &&
      (w[-5] == (words+offset7)[1]) &&
      (w[-6] == (words+offset7)[0]) &&
               !(words+offset7)[7]
     )
    {/* нашлось вхождение. */}
  }
  w++;
}
```

На этом слайде представлено вычислительное ядро, которое ищет префиксы от шаблонов.

(*) В рамках этой работы был написан генератор, который генерирует такие вычислительные ядра.
(*) Префиксы имеют длины 3 и 7.
(*) Перед этим циклом был обработан начальный кусок текста длиной 6 на предмет вхождения шаблонов длины 3.
(*) цикл начинает работу с 7 символа и проходит по всему тексту.
(*) w - указатель на текст.

Вычисление хэша по очередной букве

Релизация вычислительного ядра

```
for(unsigned i=0;i<textlen-6;i++) {
 hash404(h,w);
 if( ffdm_checkhash(h[2]) ) {
   unsigned offset3=ffdm_get_offset(h[2]);
   if(/*проверяем совпадение букв*/
      (w[ 0] == (words+offset3)[2]) &&
      (w[-1] == (words+offset3)[1]) &&
      (w[-2] == (words+offset3)[0]) &&
               !(words+offset3)[3]
    )
   {/* нашлось вхождение. */}
 }
 if( ffdm_checkhash(h[6]) ) {
   unsigned offset7=ffdm_get_offset(h[6]);
   if(
      (w[ 0] == (words+offset7)[6]) &&
      (w[-1] == (words+offset7)[5]) &&
      (w[-2] == (words+offset7)[4]) &&
      (w[-3] == (words+offset7)[3]) &&
      (w[-4] == (words+offset7)[2]) &&
      (w[-5] == (words+offset7)[1]) &&
      (w[-6] == (words+offset7)[0]) &&
               !(words+offset7)[7]
     )
    {/* нашлось вхождение. */}
  }
  w++;
}
```

На этом слайде представлено вычислительное ядро, которое ищет префиксы от шаблонов.

(*) В рамках этой работы был написан генератор, который генерирует такие вычислительные ядра.
(*) Префиксы имеют длины 3 и 7.
(*) Перед этим циклом был обработан начальный кусок текста длиной 6 на предмет вхождения шаблонов длины 3.
(*) цикл начинает работу с 7 символа и проходит по всему тексту.
(*) w - указатель на текст.

Вычисление хэша по очередной букве

Проверки на принадлежность `h[2]` и `h[6]` к {hash404(P_1),..,hash404(P_n)} при помощи FFDM.

Релизация вычислительного ядра

```
for(unsigned i=0;i<textlen-6;i++) {
 hash404(h,w);
 if( ffdm_checkhash(h[2]) ) {
   unsigned offset3=ffdm_get_offset(h[2]);
   if(/*проверяем совпадение букв*/
      (w[ 0] == (words+offset3)[2]) &&
      (w[-1] == (words+offset3)[1]) &&
      (w[-2] == (words+offset3)[0]) &&
               !(words+offset3)[3]
    )
   {/* нашлось вхождение. */}
 }
 if( ffdm_checkhash(h[6]) ) {
   unsigned offset7=ffdm_get_offset(h[6]);
   if(
      (w[ 0] == (words+offset7)[6]) &&
      (w[-1] == (words+offset7)[5]) &&
      (w[-2] == (words+offset7)[4]) &&
      (w[-3] == (words+offset7)[3]) &&
      (w[-4] == (words+offset7)[2]) &&
      (w[-5] == (words+offset7)[1]) &&
      (w[-6] == (words+offset7)[0]) &&
               !(words+offset7)[7]
     )
    {/* нашлось вхождение. */}
  }
  w++;
}
```

На этом слайде представлено вычислительное ядро, которое ищет префиксы от шаблонов.

(*) В рамках этой работы был написан генератор, который генерирует такие вычислительные ядра.
(*) Префиксы имеют длины 3 и 7.
(*) Перед этим циклом был обработан начальный кусок текста длиной 6 на предмет вхождения шаблонов длины 3.
(*) цикл начинает работу с 7 символа и проходит по всему тексту.
(*) w - указатель на текст.

Вычисление хэша по очередной букве

Проверки на принадлежность `h[2]` и `h[6]` к {hash404(P_1),..,hash404(P_n)} при помощи FFDM.

В случае успеха извлекается указатель на шаблон, используя технику FFDM.

Релизация вычислительного ядра

```
for(unsigned i=0;i<textlen-6;i++) {
 hash404(h,w);
 if( ffdm_checkhash(h[2]) ) {
   unsigned offset3=ffdm_get_offset(h[2]);
   if(/*проверяем совпадение букв*/
      (w[ 0] == (words+offset3)[2]) &&
      (w[-1] == (words+offset3)[1]) &&
      (w[-2] == (words+offset3)[0]) &&
              !(words+offset3)[3]
    )
   {/* нашлось вхождение. */}
 }
 if( ffdm_checkhash(h[6]) ) {
   unsigned offset7=ffdm_get_offset(h[6]);
   if(
      (w[ 0] == (words+offset7)[6]) &&
      (w[-1] == (words+offset7)[5]) &&
      (w[-2] == (words+offset7)[4]) &&
      (w[-3] == (words+offset7)[3]) &&
      (w[-4] == (words+offset7)[2]) &&
      (w[-5] == (words+offset7)[1]) &&
      (w[-6] == (words+offset7)[0]) &&
              !(words+offset7)[7]
     )
    {/* нашлось вхождение. */}
  }
  w++;
}
```

На этом слайде представлено вычислительное ядро, которое ищет префиксы от шаблонов.

(*) В рамках этой работы был написан генератор, который генерирует такие вычислительные ядра.
(*) Префиксы имеют длины 3 и 7.
(*) Перед этим циклом был обработан начальный кусок текста длиной 6 на предмет вхождения шаблонов длины 3.
(*) цикл начинает работу с 7 символа и проходит по всему тексту.
(*) w - указатель на текст.

Вычисление хэша по очередной букве

Проверки на принадлежность `h[2]` и `h[6]` к {hash404(P_1),..,hash404(P_n)} при помощи FFDM.

В случае успеха извлекается указатель на шаблон, используя технику FFDM.

Проверка совпадения подстроки и шаблона.

Релизация вычислительного ядра

```
for(unsigned i=0;i<textlen-6;i++) {
 hash404(h,w);
 if( ffdm_checkhash(h[2]) ) {
   unsigned offset3=ffdm_get_offset(h[2]);
   if(/*проверяем совпадение букв*/
      (w[ 0] == (words+offset3)[2]) &&
      (w[-1] == (words+offset3)[1]) &&
      (w[-2] == (words+offset3)[0]) &&
               !(words+offset3)[3]
    )
   {/* нашлось вхождение. */}
 }
 if( ffdm_checkhash(h[6]) ) {
   unsigned offset7=ffdm_get_offset(h[6]);
   if(
      (w[ 0] == (words+offset7)[6]) &&
      (w[-1] == (words+offset7)[5]) &&
      (w[-2] == (words+offset7)[4]) &&
      (w[-3] == (words+offset7)[3]) &&
      (w[-4] == (words+offset7)[2]) &&
      (w[-5] == (words+offset7)[1]) &&
      (w[-6] == (words+offset7)[0]) &&
               !(words+offset7)[7]
    )
    {/* нашлось вхождение. */}
  }
  w++;
}
```

На этом слайде представлено вычислительное ядро, которое ищет префиксы от шаблонов.

(*) В рамках этой работы был написан генератор, который генерирует такие вычислительные ядра.
(*) Префиксы имеют длины 3 и 7.
(*) Перед этим циклом был обработан начальный кусок текста длиной 6 на предмет вхождения шаблонов длины 3.
(*) цикл начинает работу с 7 символа и проходит по всему тексту.
(*) w - указатель на текст.

Вычисление хэша по очередной букве

Проверки на принадлежность `h[2]` и `h[6]` к {hash404(P_1),..,hash404(P_n)} при помощи FFDM.

В случае успеха извлекается указатель на шаблон, используя технику FFDM.

Проверка совпадения подстроки и шаблона.

Если произошло совпадение, то найдено вхождение, сохраняется позиция вхождения.

Релизация вычислительного ядра

```
for(unsigned i=0;i<textlen-6;i++) {
 hash404(h,w);
 if( ffdm_checkhash(h[2]) ) {
   unsigned offset3=ffdm_get_offset(h[2]);
   if(/*проверяем совпадение букв*/
      (w[ 0] == (words+offset3)[2]) &&
      (w[-1] == (words+offset3)[1]) &&
      (w[-2] == (words+offset3)[0]) &&
               !(words+offset3)[3]
    )
   {/* нашлось вхождение. */}
 }
 if( ffdm_checkhash(h[6]) ) {
   unsigned offset7=ffdm_get_offset(h[6]);
   if(
      (w[ 0] == (words+offset7)[6]) &&
      (w[-1] == (words+offset7)[5]) &&
      (w[-2] == (words+offset7)[4]) &&
      (w[-3] == (words+offset7)[3]) &&
      (w[-4] == (words+offset7)[2]) &&
      (w[-5] == (words+offset7)[1]) &&
      (w[-6] == (words+offset7)[0]) &&
               !(words+offset7)[7]
    )
    {/* нашлось вхождение. */}
  }
  w++;
}
```

На этом слайде представлено вычислительное ядро, которое ищет префиксы от шаблонов.

(*) В рамках этой работы был написан генератор, который генерирует такие вычислительные ядра.
(*) Префиксы имеют длины 3 и 7.
(*) Перед этим циклом был обработан начальный кусок текста длиной 6 на предмет вхождения шаблонов длины 3.
(*) цикл начинает работу с 7 символа и проходит по всему тексту.
(*) w - указатель на текст.

Вычисление хэша по очередной букве

Проверки на принадлежность `h[2]` и `h[6]` к {hash404(P_1),..,hash404(P_n)} при помощи FFDM.

В случае успеха извлекается указатель на шаблон, используя технику FFDM.

Проверка совпадения подстроки и шаблона.

Если произошло совпадение, то найдено вхождение, сохраняется позиция вхождения.

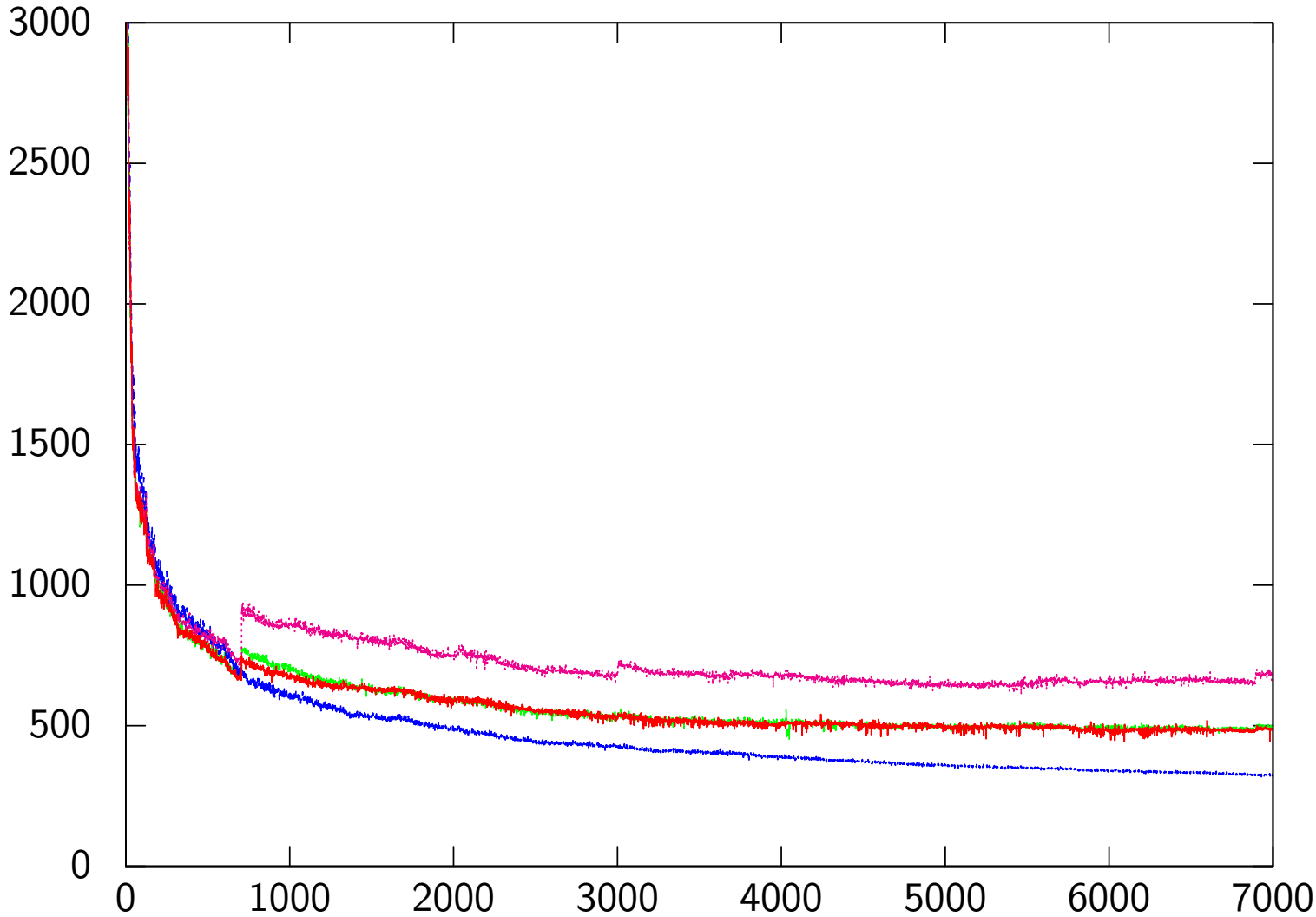
После прохода по тексту для найденных префиксов будут проверяться суффиксы.

Сравнение производительности

Измерения производительности

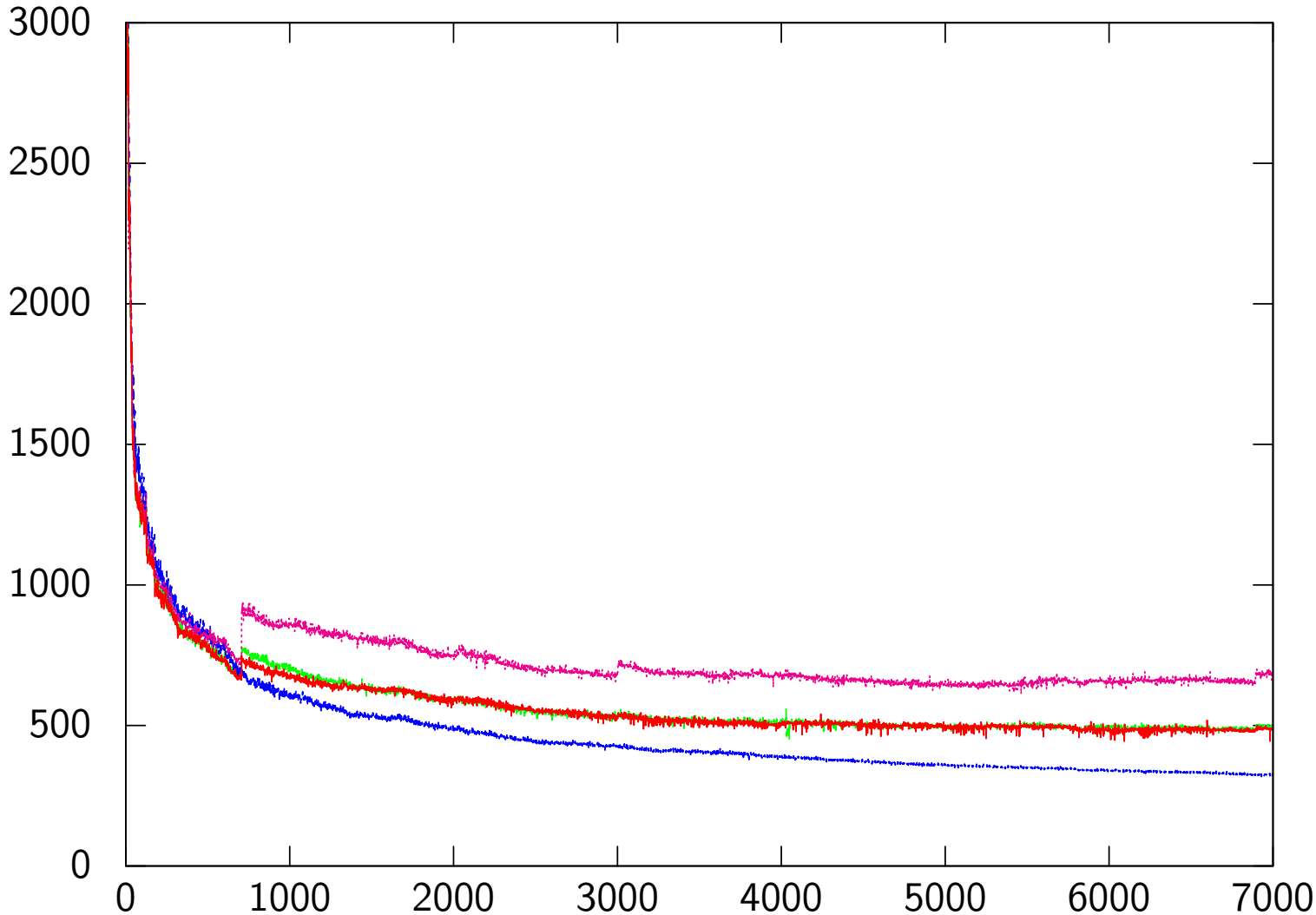
- Производительность будет замеряться на 4х файлах
  - William Shakespeare - Romeo and Juliet
  - James Jones - From here to eternity
  - Samuel Shem - Mount mistery
  - Salinger - The catcher in the rye
- Интерес представляет зависимость производительности реализации от количества шаблонов.
- На графиках далее по оси X будет откладываться количество шаблонов, а по Y: $\frac{\text{size(text)}}{\text{time}}$ — Megabit/Sec
- В замерах не учитывается время затраченное на построение поисковых структур(автоматов, trie,...)

Измерения производительности



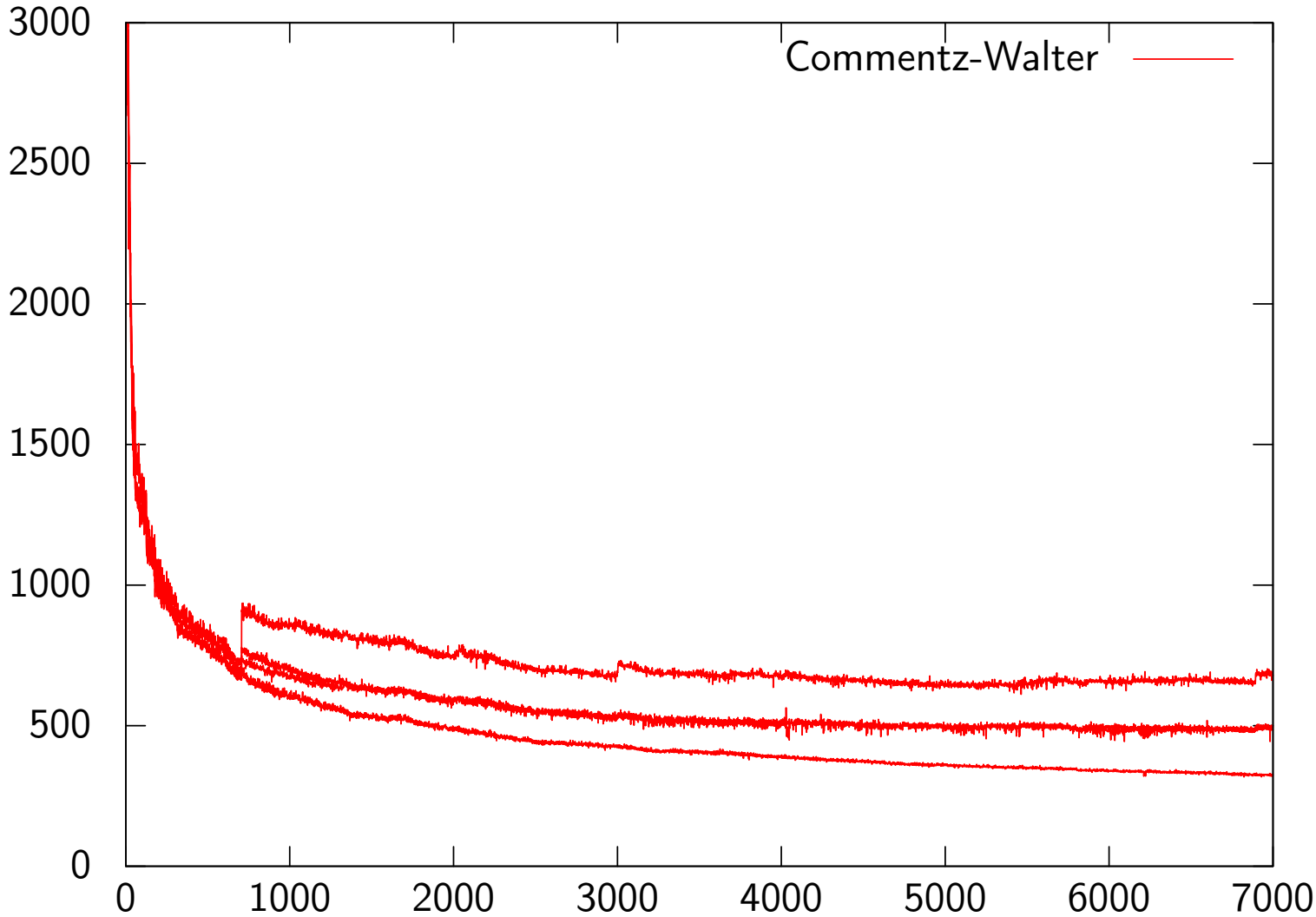
Представленные кривые отображают производительность алгоритма Commentz-Walter(реализация GNU grep).

Измерения производительности



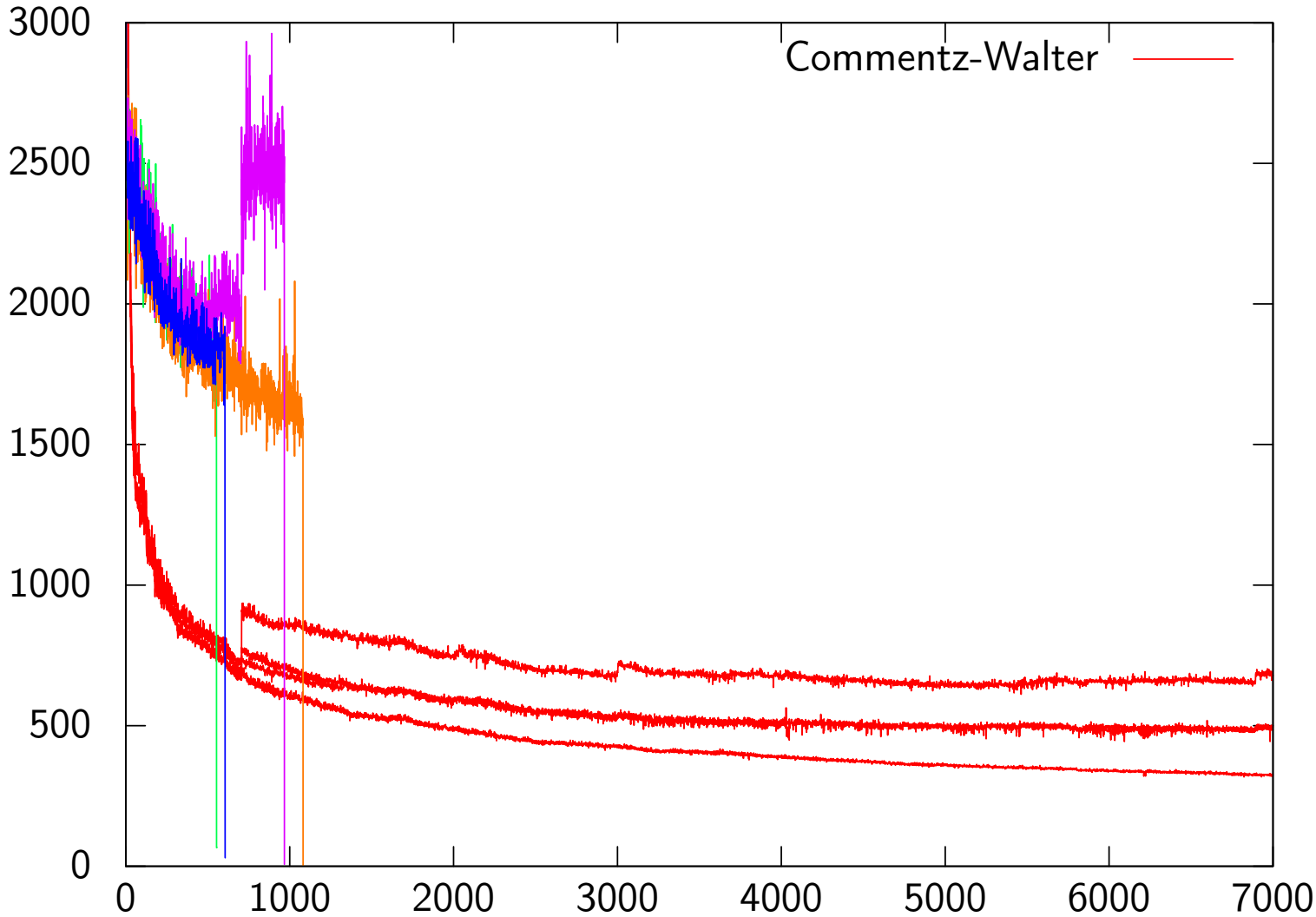
Окрасим группу этих кривых в красный цвет.

Измерения производительности



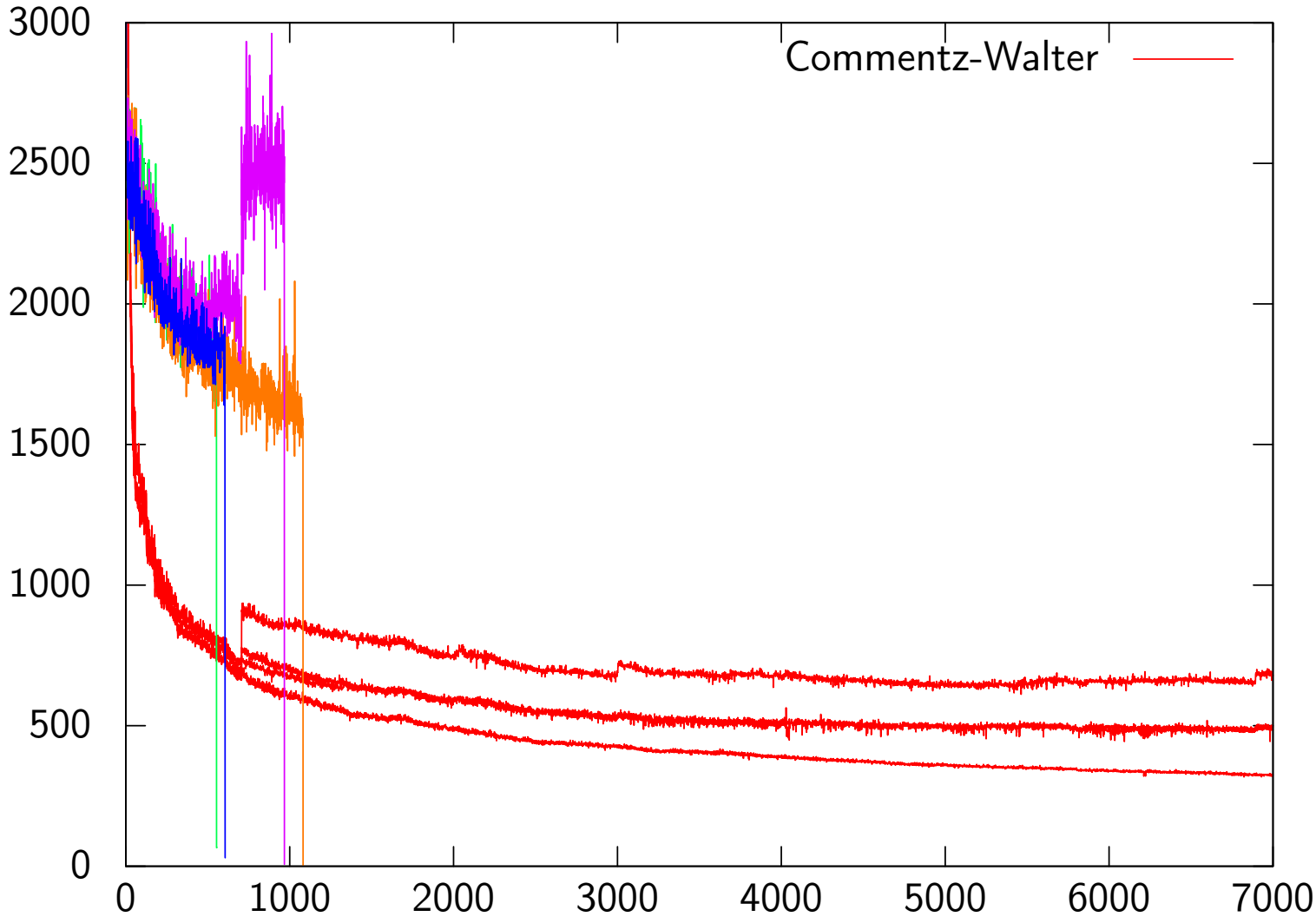
Окрасим группу этих кривых в красный цвет.

Измерения производительности



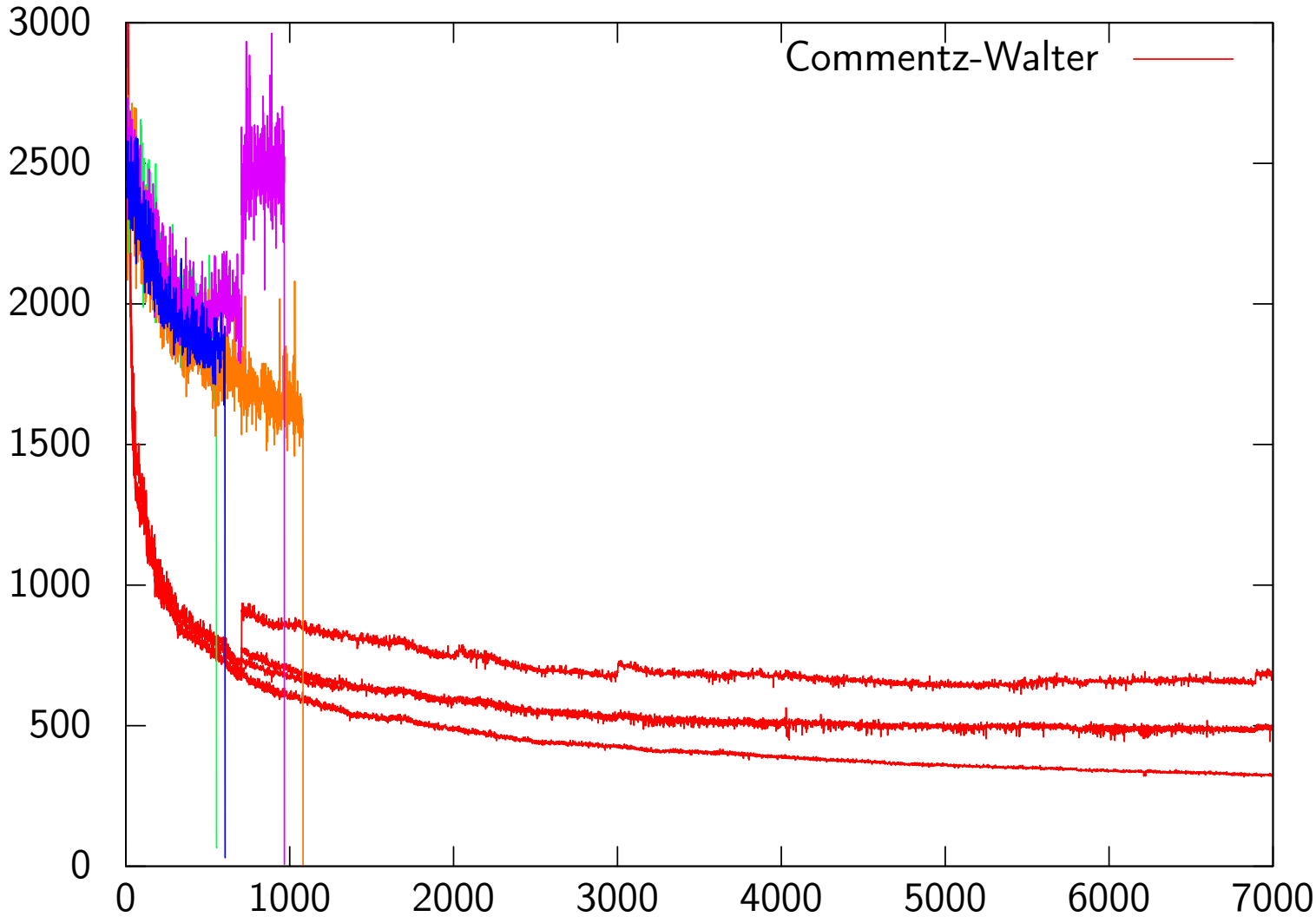
Добавленные кривые иллюстрируют производительность DFA(реализация GNU grep).

Измерения производительности



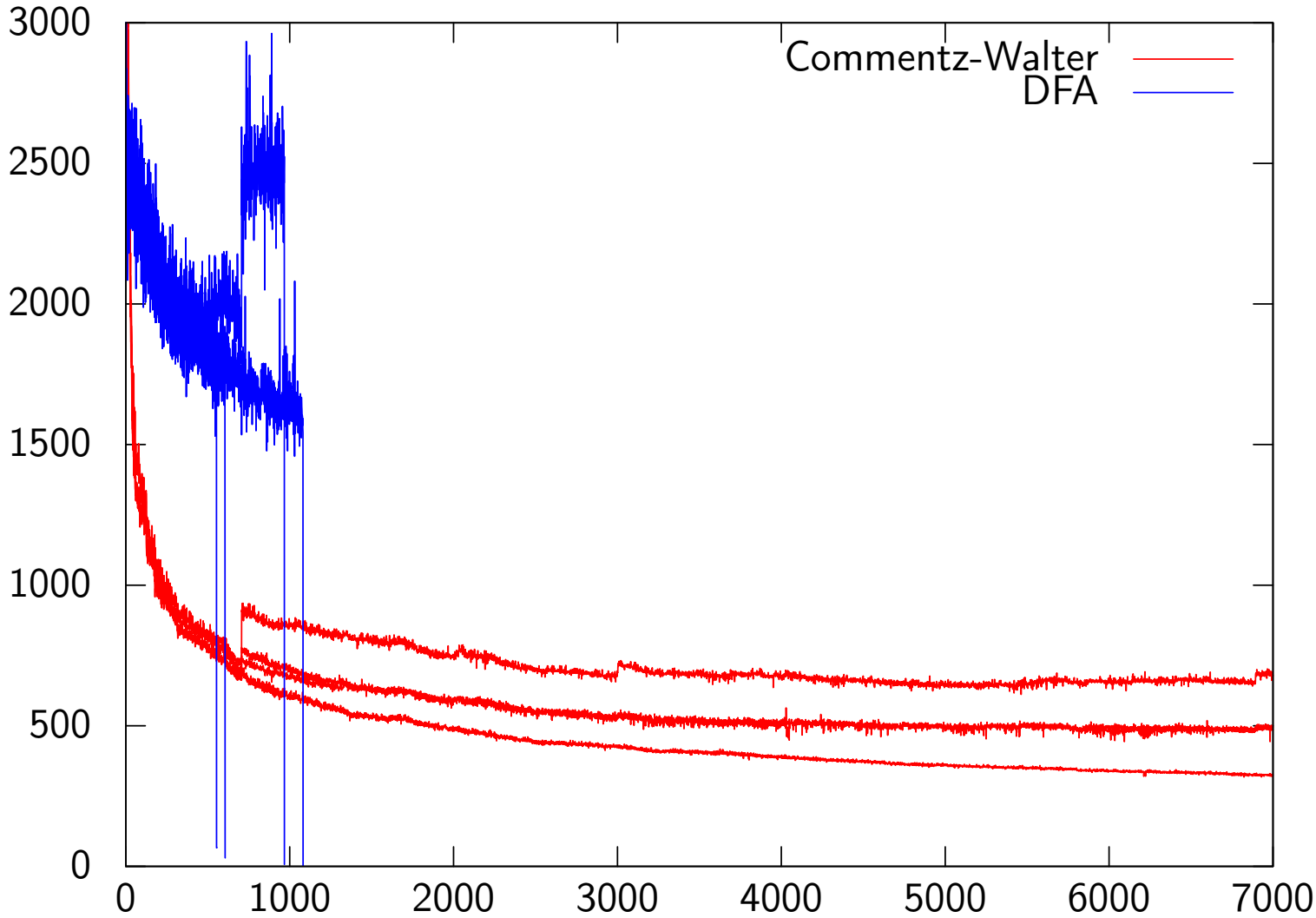
Вертикальные обрывы — это спады производительности, о которых говорилось ранее.

Измерения производительности



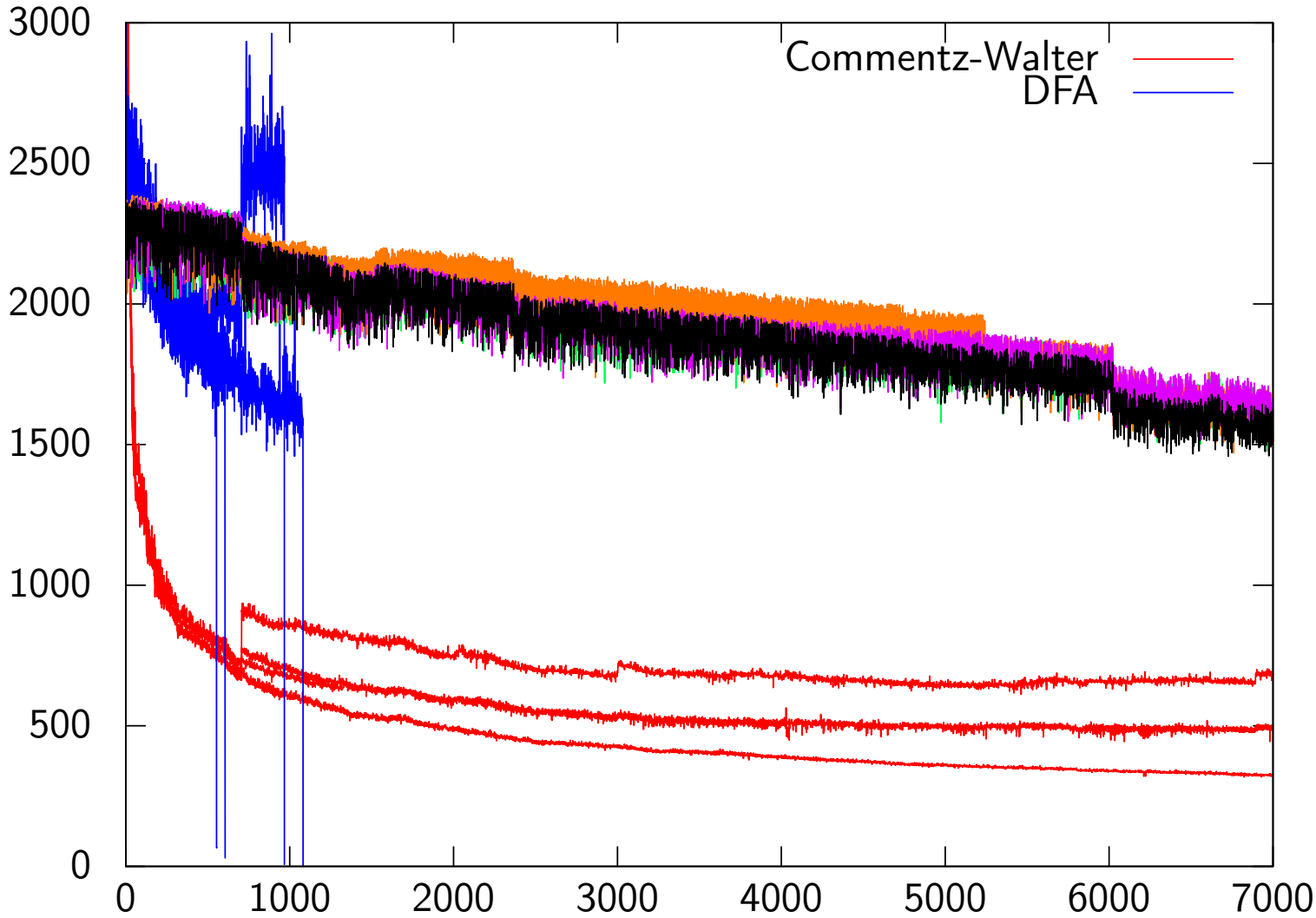
Окрасим группу кривых с производительностью DFA в синий цвет.

Измерения производительности



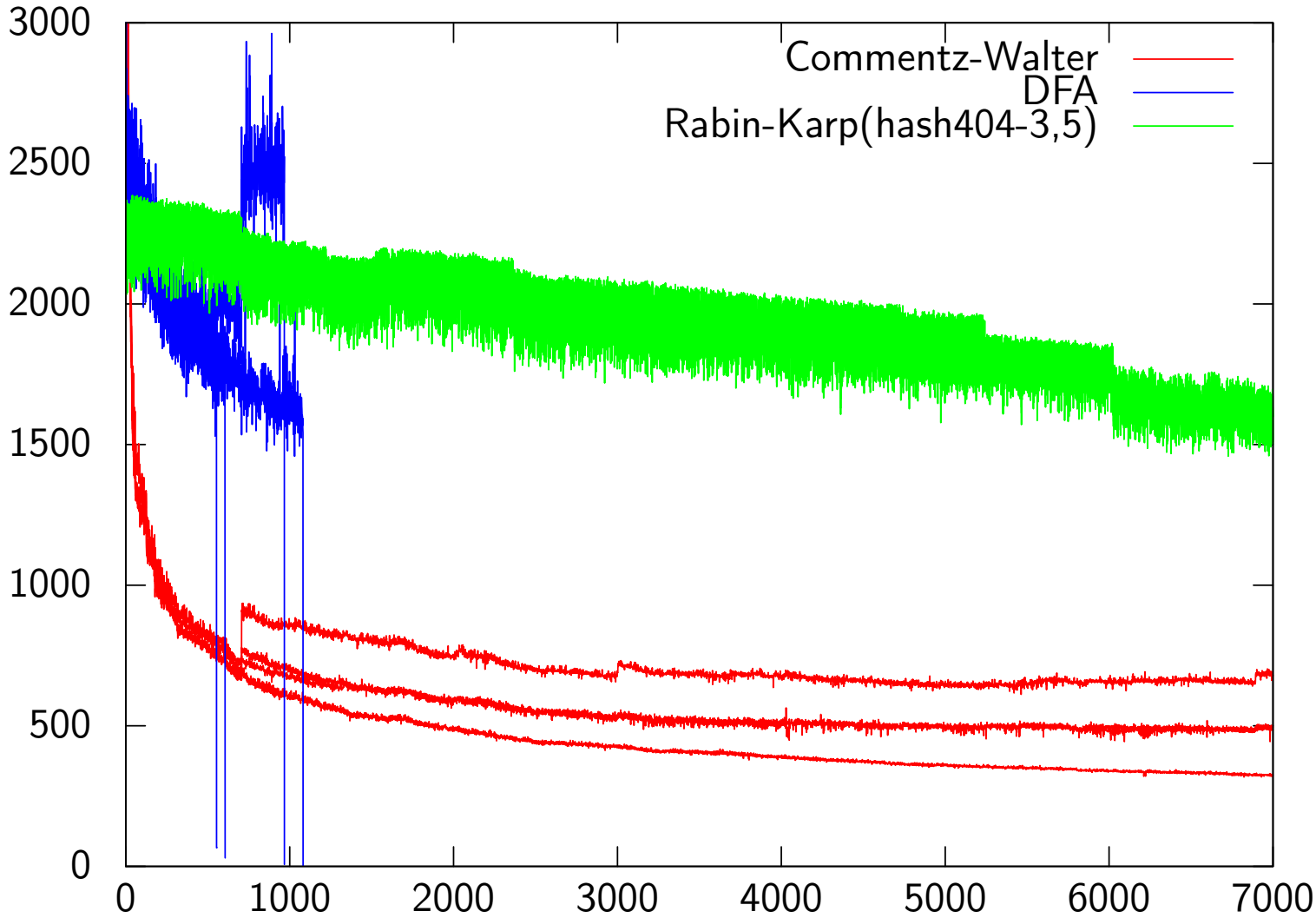
Окрасим группу кривых с производительностью DFA в синий цвет.

Измерения производительности



Эта группа кривых демонстрирует производительность алгоритма Rabin-Karp(реализация – авторы доклада, hash404)

Измерения производительности



hash404 обгоняет Commentz-Walter примерно в 2–3 раза.

Перебор параметров для hash404

Какие брать параметры ({m_i},mask) ?

hash404(a_1..a_s)=((((..((a_1<<m_1)^a_2)..)<<m_{s-1})^a_s) & mask

(+) FFDM $\Rightarrow \forall i \neq j$ hash404(P_i) $\neq$ hash404(P_j)

Было замечено, что если шаблонов > 2000, то из (+) $\Rightarrow$

1. $\sum m_i + 8 =$ mask. Т.е. применять маску нет смысла.
2. Если $\sum m_i + 8$ мала, то скорее всего, инъективности не будет.

$$\Downarrow$$

(++) $N \leq \sum m_i + 8 \leq$ sizeof(unsigned) $= 32$.

(например, для 7000 слов N=28)

Учитывая (+),(++) перебор можно организовать так:

- Для всякого L$\in [N, 32]$ нужно рассмотреть разбиение на все возможные суммы $\sum_{i=1}^{l_s-1} m_i =$L. Каждое разбиение проверить на свойство (+)
- Проверку (+) можно организовать при помощи хэш-множества с разрешением коллизий методом цепочек. Перед добавлением hash404(P_i) в множество проверяем, есть ли там это число. Если есть, то (+) нарушено.

Перебор параметров для hash404

```
(*) if (h_k ∈ hashes)
(**)  if (T[j-l_k+1,..,j] ∈ subsP)
        print j, T[j-l_k+1,..,j]
```

Добавление (*) ускоряет программу в среднем на 30%

Также при плохих ({m_i},mask) наблюдается фатальный спад производительности(до 500Mbit/sec) из-за того, что проверка часто пропускает исполнение к (**). Без проверки спада нет.

Плохие ({m_i},mask) попадаются, если они подбираются для <400 шаблонов.

Этого можно избежать очень просто. Можно добавить к q шаблонам(q<400), например, 2000 каких-нибудь шаблонов, и подобрать параметры для 2000+q шаблонов. И для поиска искомых q шаблонов использовать эти параметры.

Перебор параметров для hash404

- В рамках проекта был реализован поисковик инъективных хэшей, который умеет перебирать по суммам.
- Удалось найти инъективные хэши для 15454 шаблонов, от которых брались префиксы длины 3 и 5, и это не предел.
- Перебор всех разбиений чисел из диапазона [1..32] был сделан за 3 секунды на 4х ядрах intel core i5.

Производительность на 15454 шаблонах:

```
hash404          1272 1313 1307 1274
Commentz-Walter 262  502  708  476
```

Коллизии

Как говорилось ранее, для FFDM нужен инъективный хэш.

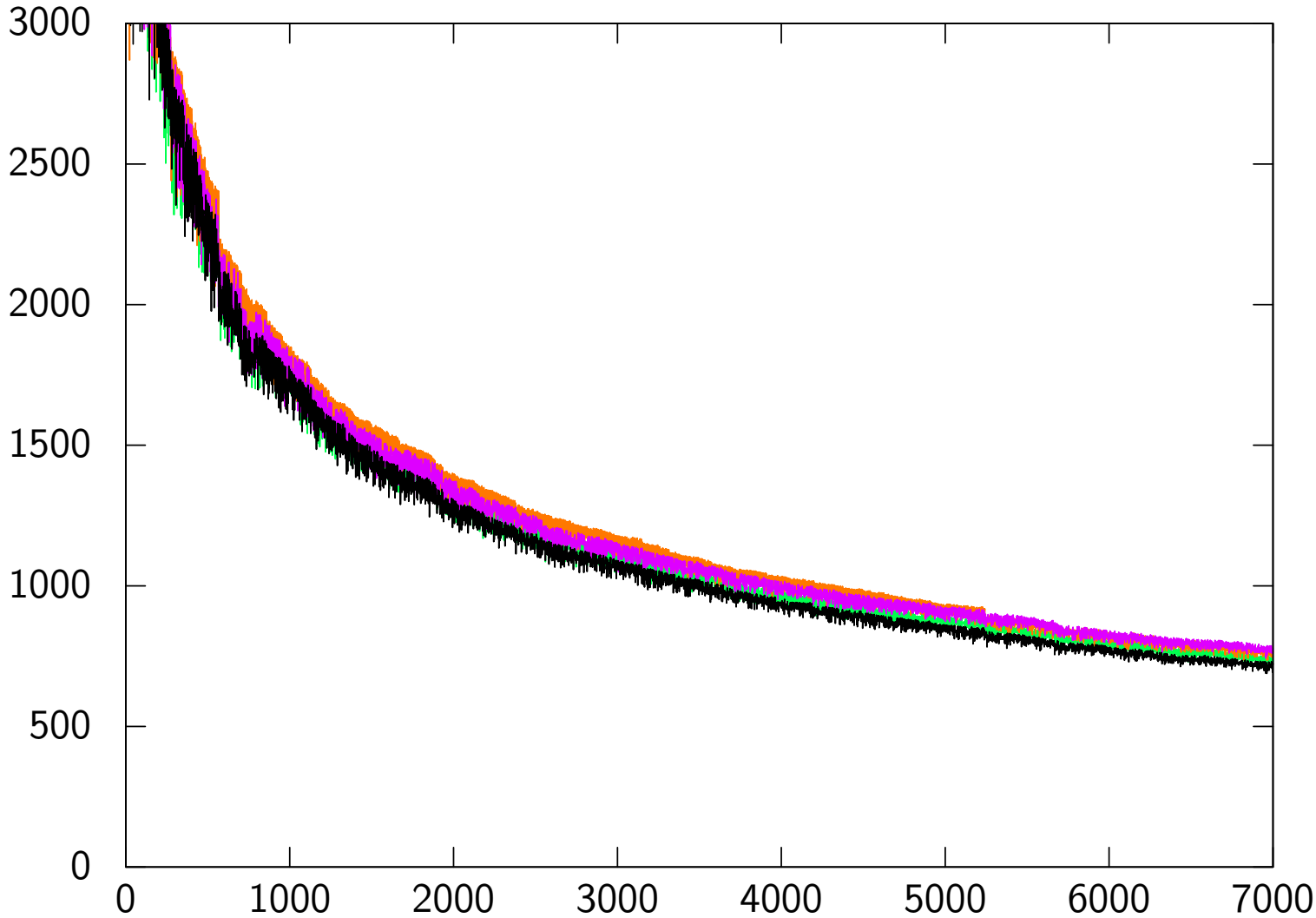
Если задавать длины префиксов d_1,..,d_p и брать от каждого шаблона максимально возможную длину, тогда могут возникнуть коллизии. И никакие напарметры не дадут инъективного хэша.

Можно поступить так.
Пусть длина самого коротного шаблона не меньше 3. Будем отделять от шаблона префиксы длиной только 3.

Тогда можно положить m1=8,m2=8
(или если все шаблоны в ASCII m1=7,m2=7).
В этом случае хэш будет в точности первые 3 байта шаблона. Такой хэш заведомо инъективен.

Коллизии

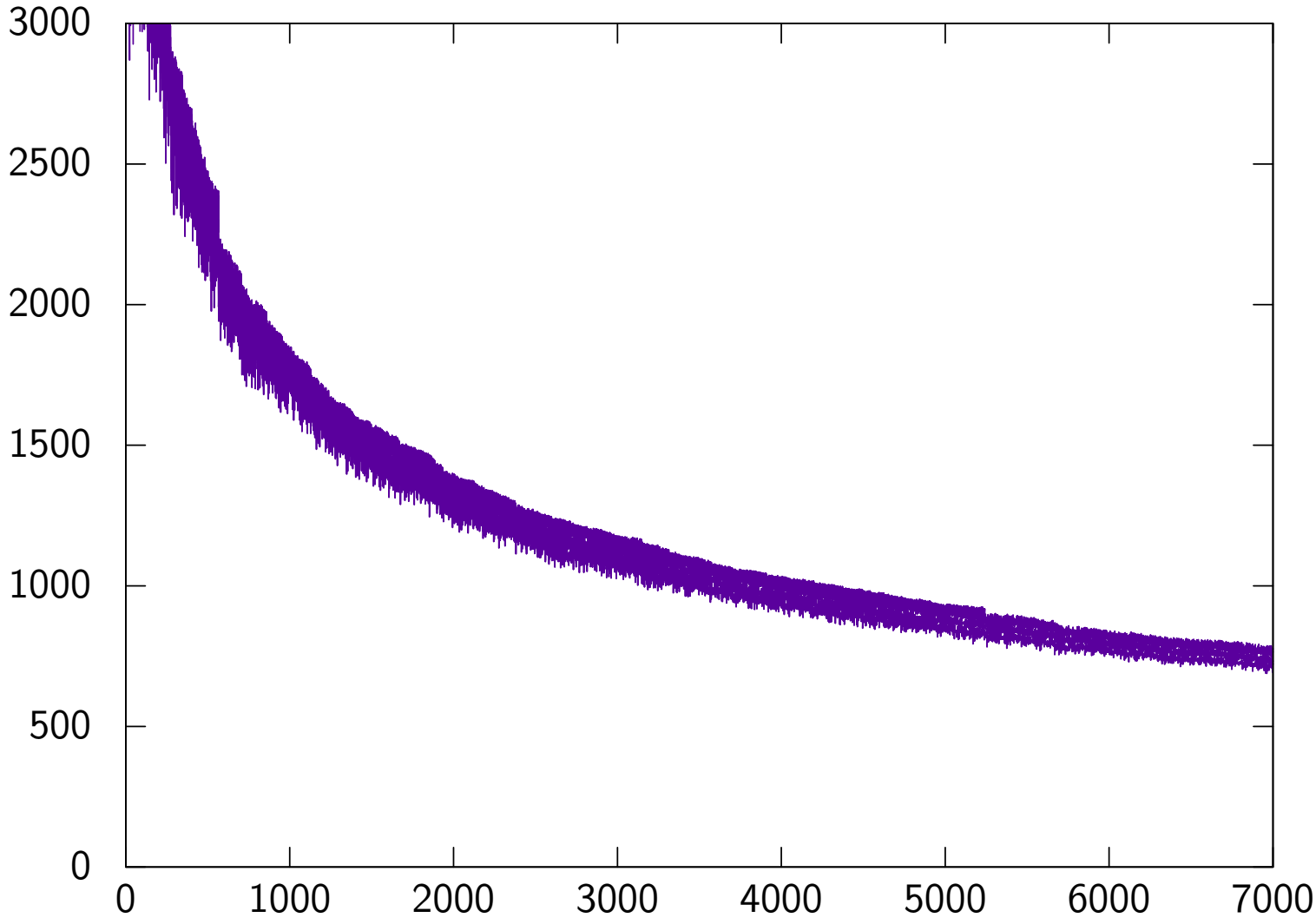
$h(a_1a_2a_3) = ((((a_1 \ll 7) \hat{} a_2) \ll 7) \hat{} a_3)$



Производительность для m_1=7,m_2=7.

Коллизии

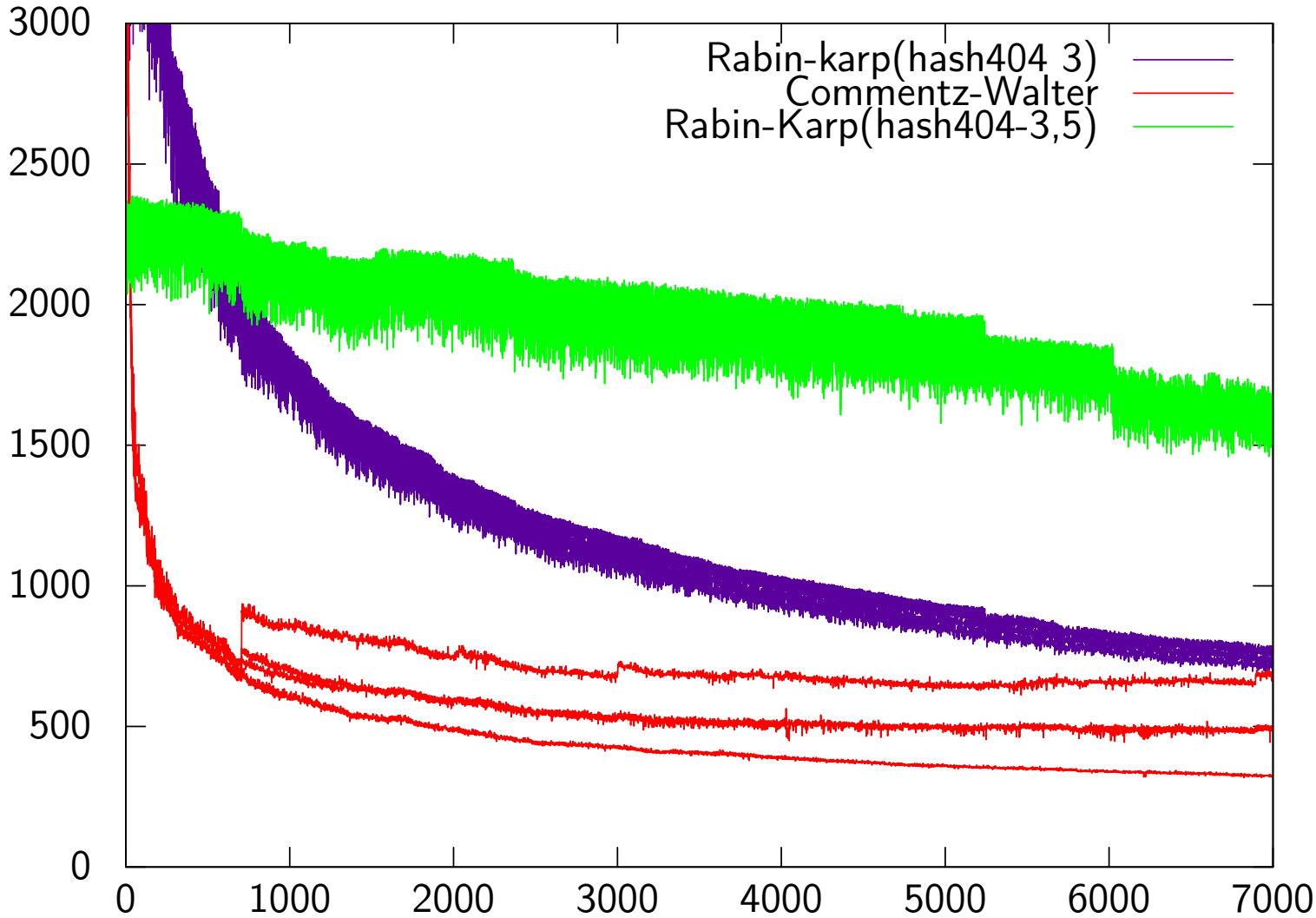
$h(a_1a_2a_3) = ((((a_1<<7)\hat{}a_2)<<7)\hat{}a_3)$



Окрасим группу кривых в фиолетовый.

Коллизии

$h(a_1a_2a_3) = ((((a_1<<7)\hat{}a_2)<<7)\hat{}a_3)$



Добавим Commentz-Walter и hash404 с префиксами 3,5.

Коллизии

Для устранения коллизий предлагается следующее:

- Если у пары основ произошла коллизия, то нужно от шаблонов брать префикс меньшей длины.
- Первые два сдвига сделать по 8 или 7.

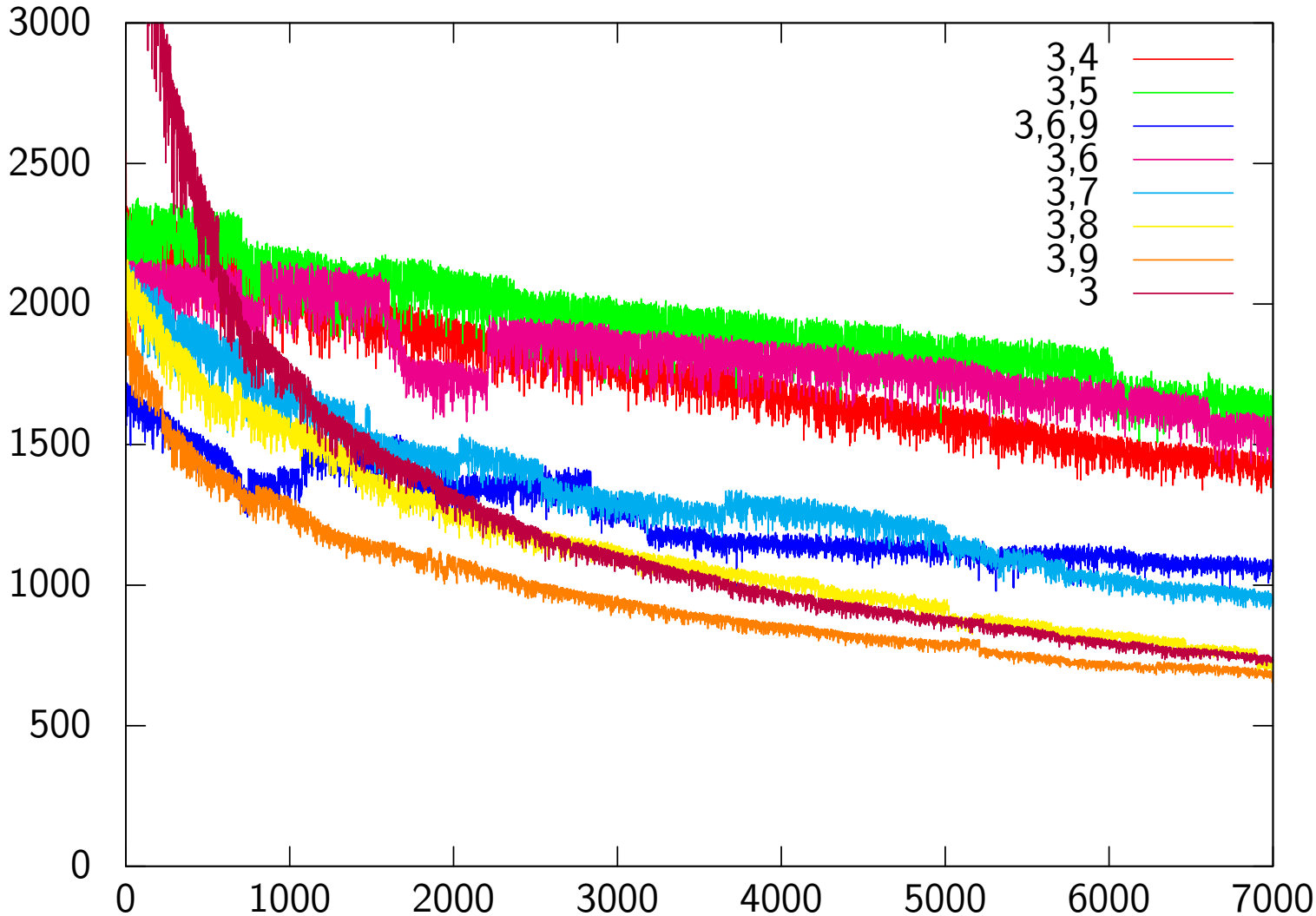
При такой миграции префиксов производительность предлагаемого алгоритма будет уменьшаться, однако, на достаточно большом количестве шаблонов, будет превосходить Commentz-Walter.

Производительность на 15454 шаблонах:

```
hash404(3,5)     1272 1313 1307 1274
Commentz-Walter 262  502  708  476
hash404(3)       502  488  515  467
```

Разные префиксы и производительность.

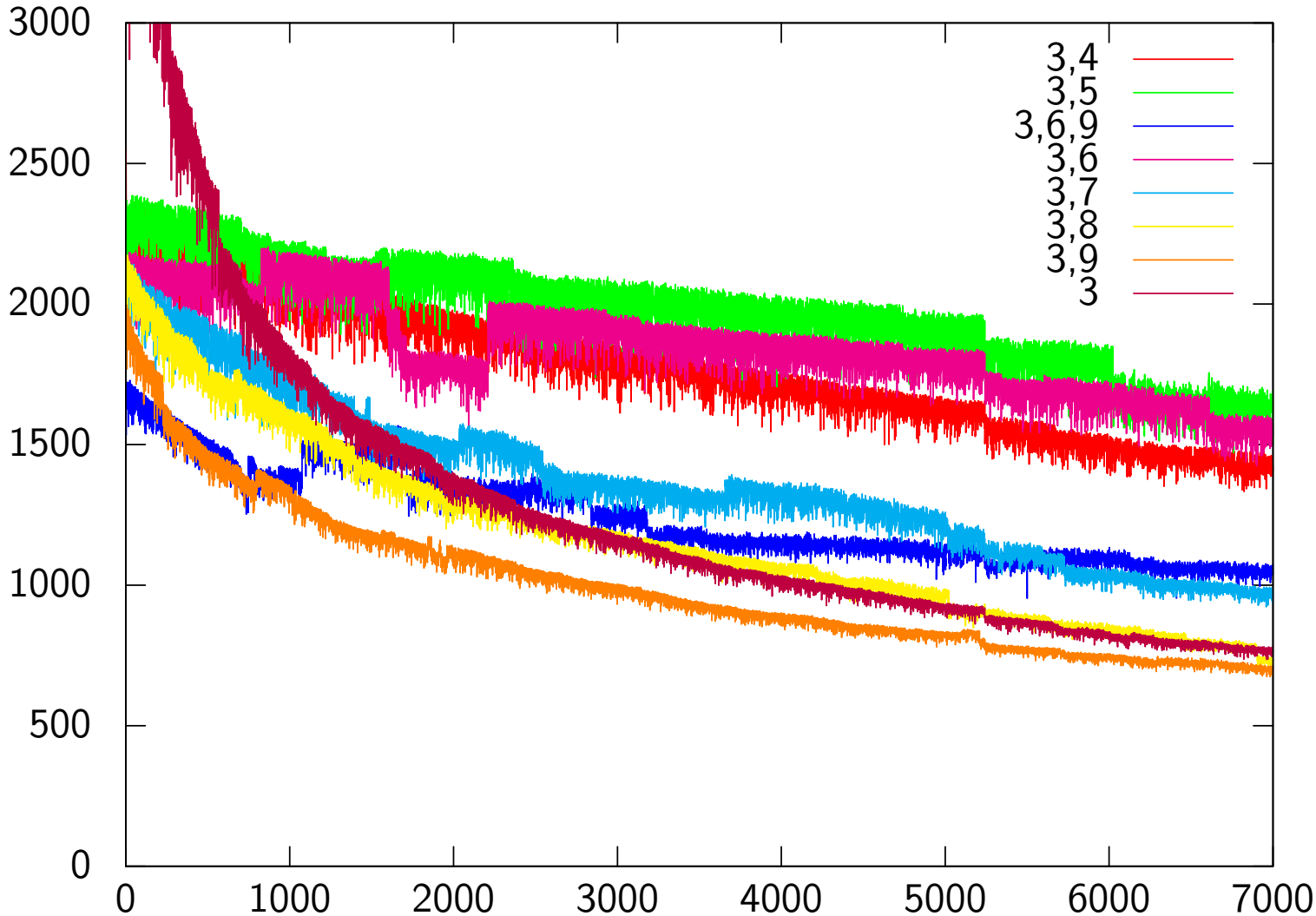
Как говорилось ранее, выбор длин префиксов — $\{d_i\}$ существенно влияет на производительность.



Файл 1

Разные префиксы и производительность.

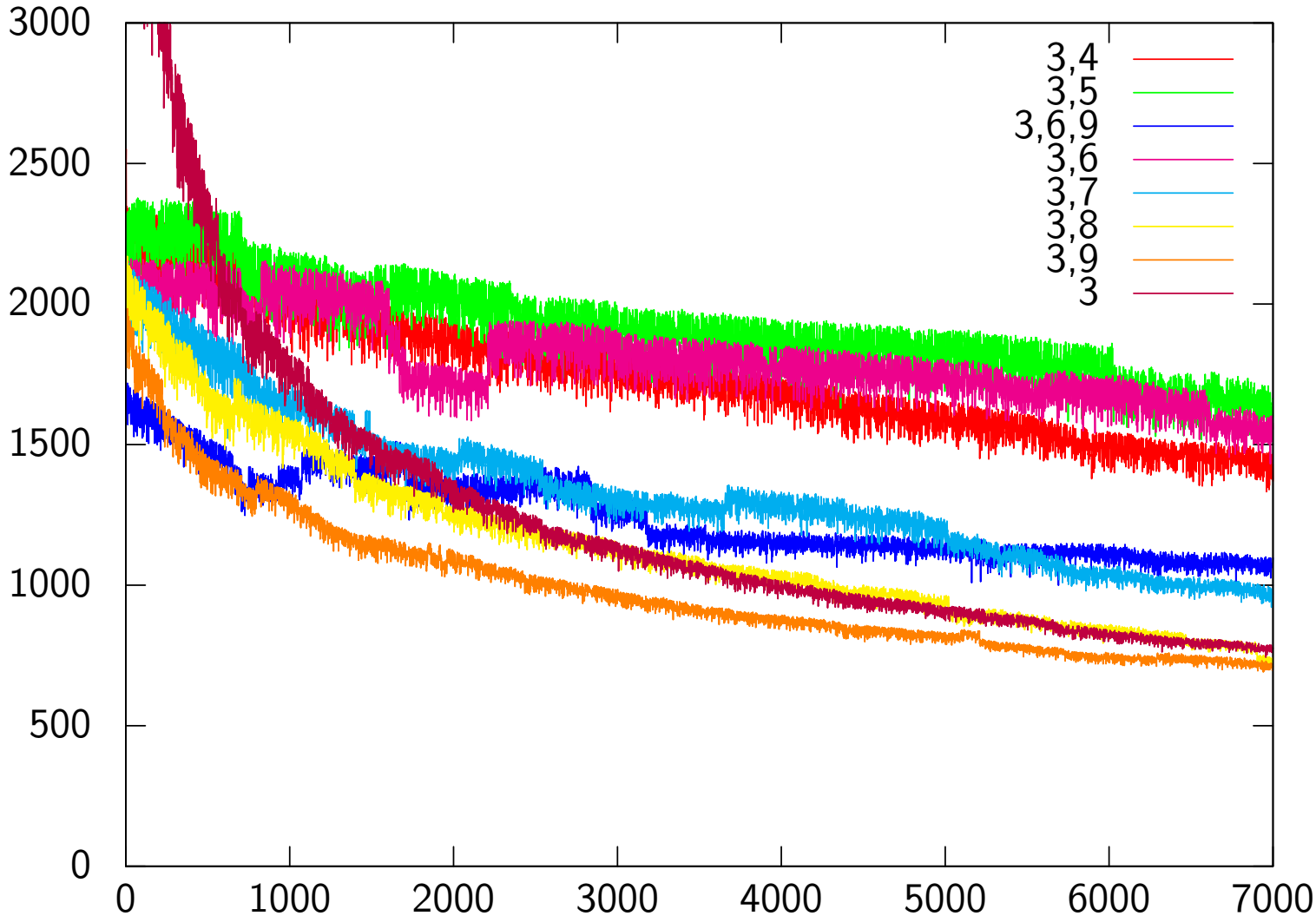
Как говорилось ранее, выбор длин префиксов — $\{d_i\}$ существенно влияет на производительность.



Файл 2

Разные префиксы и производительность.

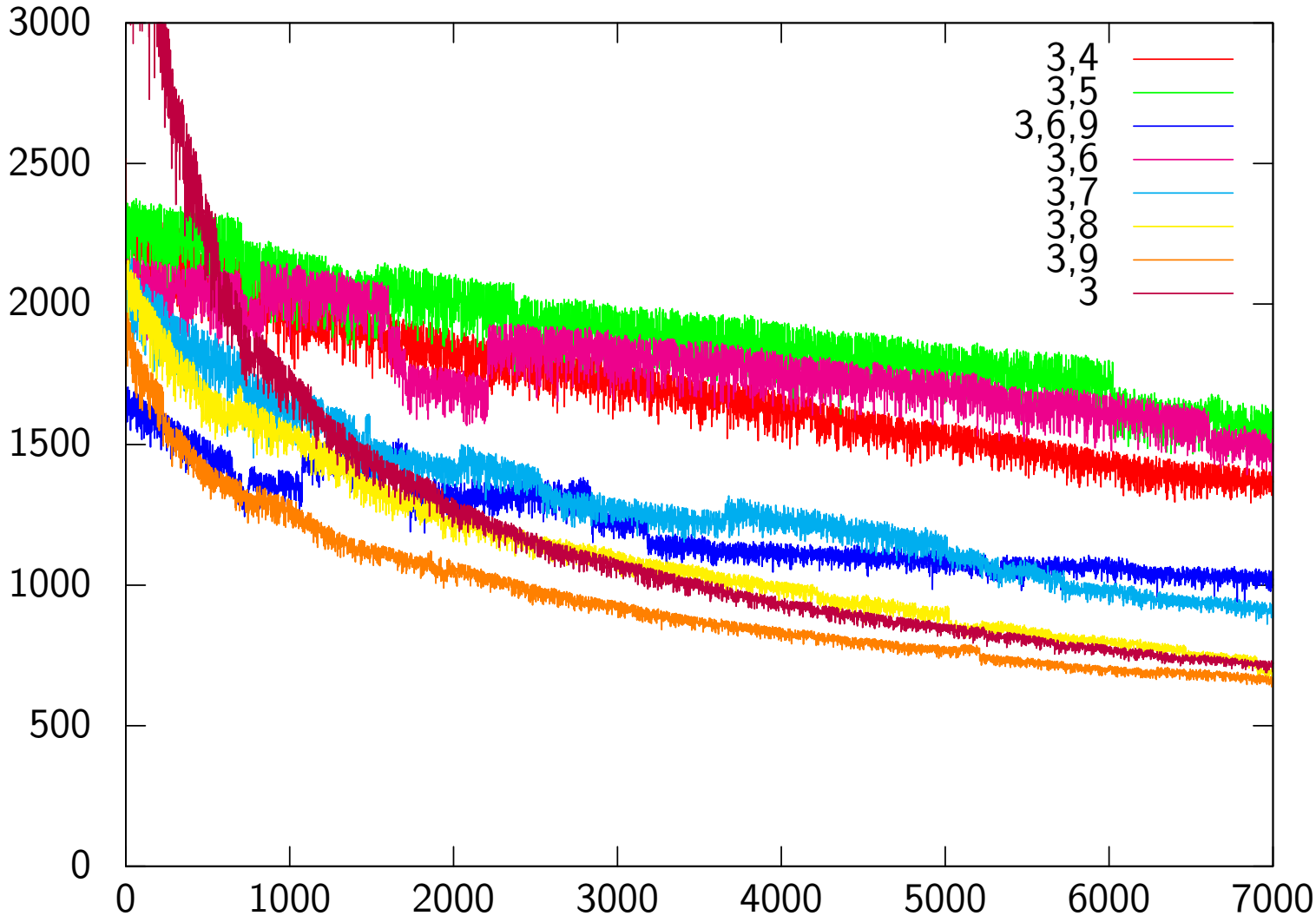
Как говорилось ранее, выбор длин префиксов — $\{d_i\}$ существенно влияет на производительность.



Файл 3

Разные префиксы и производительность.

Как говорилось ранее, выбор длин префиксов — $\{d_i\}$ существенно влияет на производительность.



Файл 4

Что было сделано

В рамках этого проекта было разработано:

- Хэш-функция, при помощи которой получилось обогнать алгоритм Commentz-Walter(GNU grep) практически в 2 раза.
- Генератор C++ кода, который генерирует вычислительные ядра по заданным шаблонам и параметрам хэш-функции.
- Поисковик инъективных хэшей для заданного множества шаблонов.

Выводы и перспективы

Перспективы, критерий хороших параметров

Хотим подобрать такие параметры $\{m_i\}$, чтобы проверки (*), (**) работали наиболее эффективным образом.

(*) `if (`$h_{d_k} \in$ `hashes)`

(**) `if (T[j-`d_k`+1,..,j] ∈ subsP)`

`print j, T[j-`d_k`+1,..,j]`

▷ Пусть $\xi_1, ..., \xi_s$ — случайные величины, выдающие слова длины $d_1, .., d_s$

▷ Пусть E_k — среднее число проверок в (**), которое отличило ξ_k от шаблонов, при условии, что $\xi_k \notin$ subsP.

▷ Пусть p_k — вероятность того, что ξ_k прошла проверку (*), при условии, $\xi_k \notin$ subsP для хэша с параметрами $\{m_i\}$.

Тогда $n(\sum_{k=1}^{s} p_k E_k P(\xi_k \notin \text{subsP}))$ — с большой вероятностью примерное число сравнений символов на $n \times k$ случайных словах с использованием hash404 и параметров $\{m_i\}$.

Можно задать некоторое вероятностное распределение и попытаться минимизировать величину $\sum_{k=1}^{s} p_k E_k P(\xi_k \notin \text{subsP})$. Возможно это даст прирост к производительсноти.

Хотим сделать:

- Для заданных шаблонов определять длины префиксов, которые дадут наибольшую производительность.
- Реализовать переброс шаблонов на префиксы меньшей длины.
- Использование SIMD.
- Реализовать вычисление вероятностного критерия.
- Генерировать код для CUDA.
- Генерировать VHDL.
- Попробовать использование стоп-символов.
- Внедрить разработки и анализировать сетевые пакеты.

Спасибо за внимание!