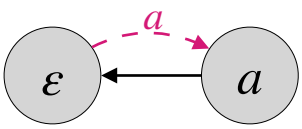
Минимален суфиксен автомат. Анализ, тестови сценарии и разработка. (Андрей Стоев)

Ще построим минимален суфиксен автомат чрез *on-line* алгоритъма на Blumer et. al., за думата $w = aabbabbbb$, като по-късно ще стане ясно защо съм избрал точно тази дума. С **q** ще означаваме текущото състояние (най-дългият суфикс – може да си го представяме като текущ показател или курсор). **Черните** ребра са родителската функция на суфиксното дърво $\mathcal{T}_w$, **розовите** ребра са преходите $\delta(u, \cdot) \nrightarrow v$, където $u, v \in \Sigma^*$, на минималния суфиксен автомат $\mathcal{A}_w$. Финалните състояния $\mathcal{F}_w$ не се поддържат от алгоритъма, за да не го утежняват и тях ще калкулираме в края му (но въпреки това ще ги броим *in-place* с цел проверка на изхода след като напишем кода). С **nq** ще отбелязваме задължителното ново състояние при добавяне на буква, с **v** ще бележим първото състояние изкачвайки се по суфиксното дърво, което е имало дефиниран преход с новата буква и потенциално може да породи второто ново състояние (други нови състояния няма да има), с **nv** ще бележим второто ново състояние. Всички останали нотации, които не са описани следват означения от лекциите на **д-р Стефан Герджиков**.

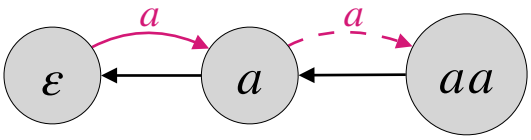
$w = \varepsilon \cdot a = a$



states: 2
trans: 1
fin: 2

фиг. 1

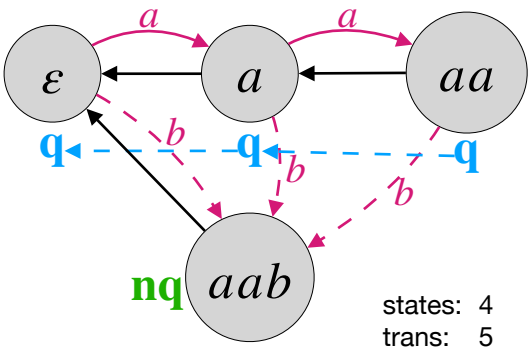
$w = w \cdot a = aa$



states: 3
trans: 2
fin: 3

фиг. 2

$w = w \cdot b = aab$

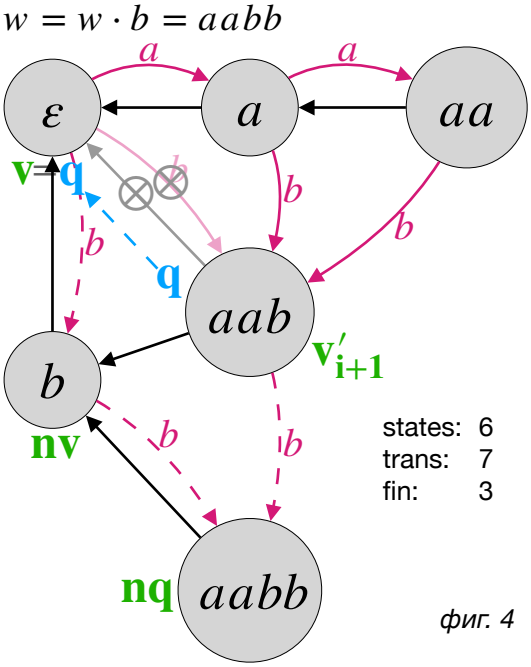


states: 4
trans: 5
fin: 2

фиг. 3

Първото задължително ново състояние ще е най-дългия суфикс след добавянето на новата буква, тъй като то ще образува нов клас на еквивалентност. $nq = aab$; $q = aa$; $\delta(q, b) = \perp \Rightarrow$ добавяме нов преход $\delta(q, b) = nq$. Траверсираме нагоре към корена на суфиксното дърво $\mathcal{T}_w$ чрез бащината му функция (черните насочени ребра). $q = p(q) = a$. Отново проверяваме за преход на q с ново добавената буква b. $\delta(q, b) = \perp \Rightarrow$ добавяме нов преход $\delta(q, b) = nq$; $q = p(q) = \varepsilon$; $\delta(q, a) = \perp \Rightarrow$ нов преход $\delta(q, a) = nq$; Алгоритъмът спира, тъй като $p(q) = -1$, т.е. текущото състояние q е корена на суфиксното дърво. Освен автоматните преходи е необходимо да поддържаме и функцията на бащите на суфиксното дърво, $p(nq) = q$. Накрая поставяме курсора на правилното му място: $q = nq$, за да го подготвим за следваща итерация.

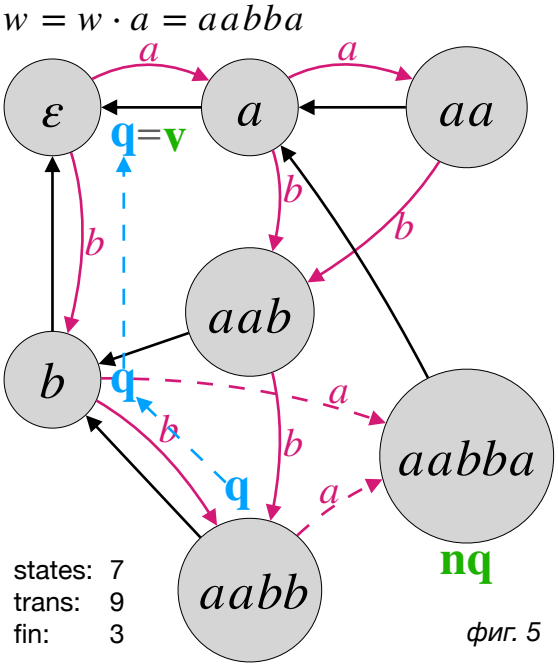
$w = w \cdot b = aabb$



states: 6
trans: 7
fin: 3

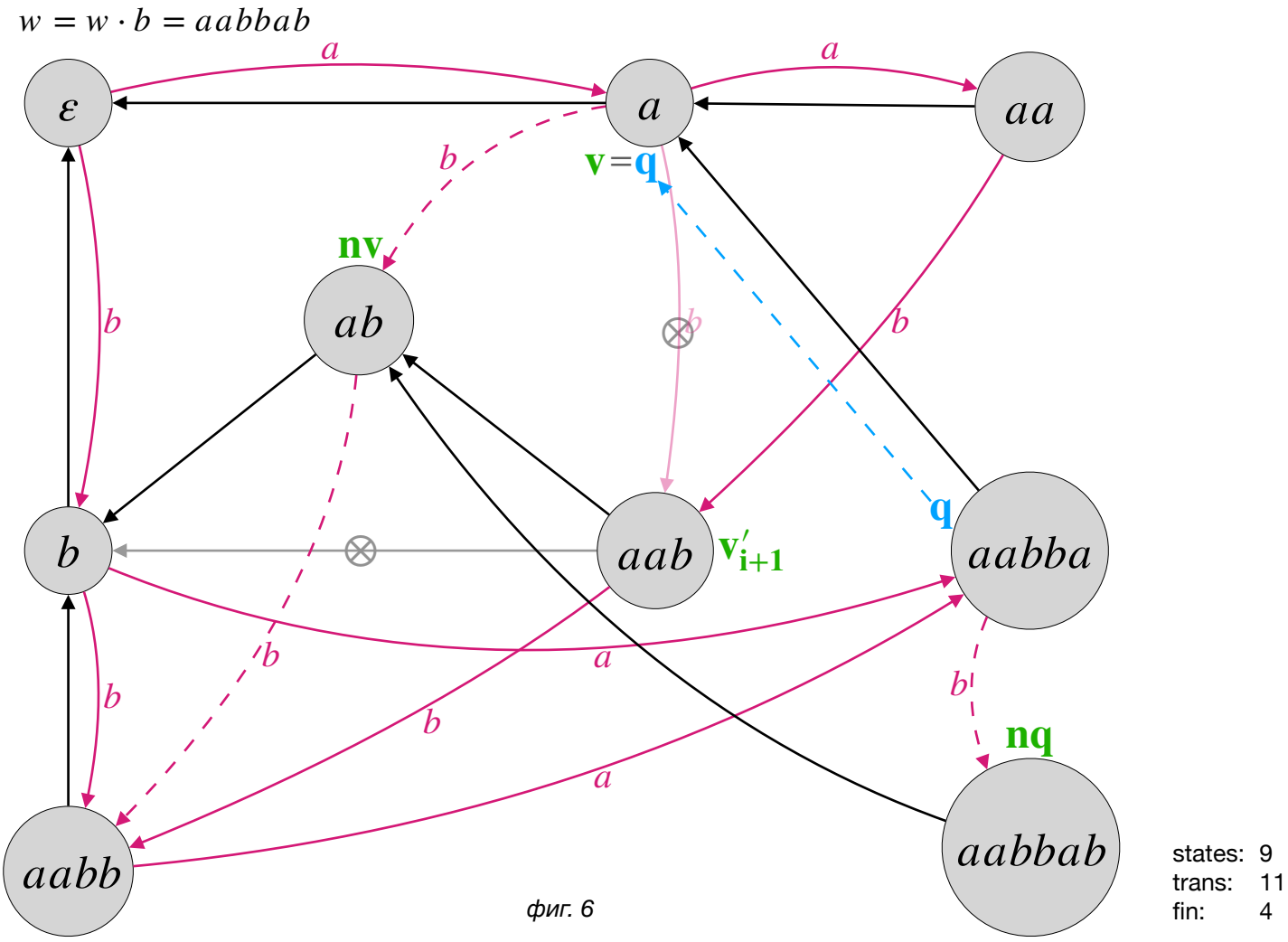
фиг. 4

Отново знаем, че $nq = aabb$; $q = aab$ и $\delta(q, b) = \perp \Rightarrow$ добавяме нов преход $\delta(q, b) = nq$; $q = p(q) = \varepsilon$; $\delta(q, b) = aab = v'_{i+1} \Rightarrow v = q$, но $v \cdot b = \varepsilon \cdot b = b \neq v'_{i+1} \Rightarrow$ създаваме ново състояние $nv = v \cdot b = b$. Новото състояние копира (наследява) всички преходи на v'_{i+1}, а преходите към nv са пренасочени преходи от v отиващи до v'_{i+1} с b и всички останали представители, които са били толкова къси, че с буквата b вече не отиват до v'_{i+1}, а отиват до новото състояние nv (тях отново може да намерим с бащината функция на $\mathcal{T}_w$).
Остана да актуализираме бащината функция на суфиксното дърво: $p(nv) = p(v'_{i+1})$; $p(n) = p(v'_{i+1}) = nv$;
И отново се подготвяме за последваща буква: $q = nq$.



фиг. 5

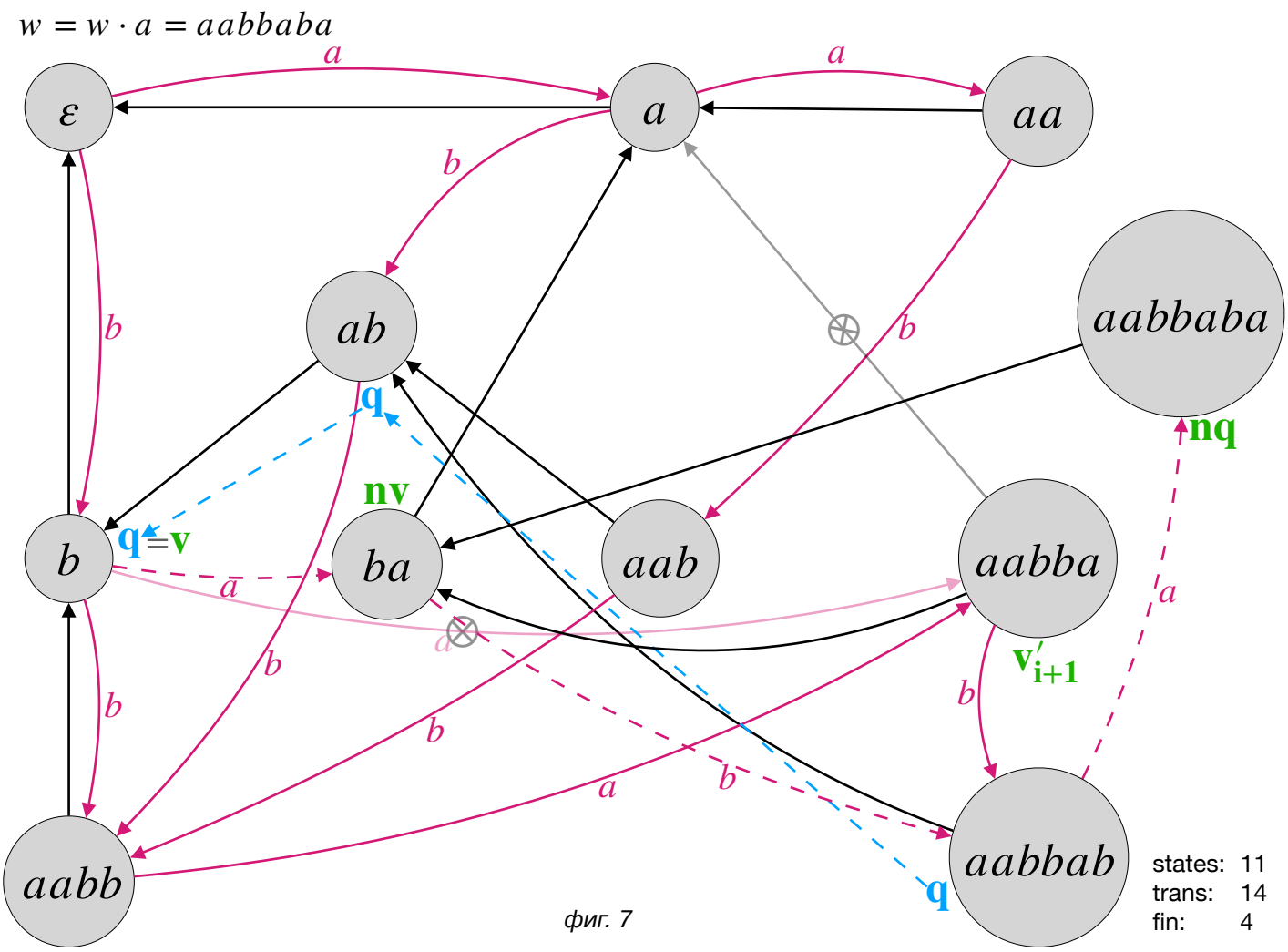
$nq = aabba;\ q = aabb$ и $\delta(q,a) = \perp \Rightarrow$ нов преход $\delta(q,a) = nq;\ q = p(q) = b;\ \delta(q,a) = \perp \Rightarrow$ нов преход $\delta(q,a) = nq;\ q = p(q) = \varepsilon$, което е коренана суфиксното дърво и началото на автомата, т.е. има преходи с всяка буква от $w \Rightarrow v = q$; $q \cdot a = \varepsilon \cdot a = a = \delta(q,a) \Rightarrow$ няма нужда от ново състояние тъй като $v'_{i+1} = nv = a = p(nq);\ q = nq$.



фиг. 6

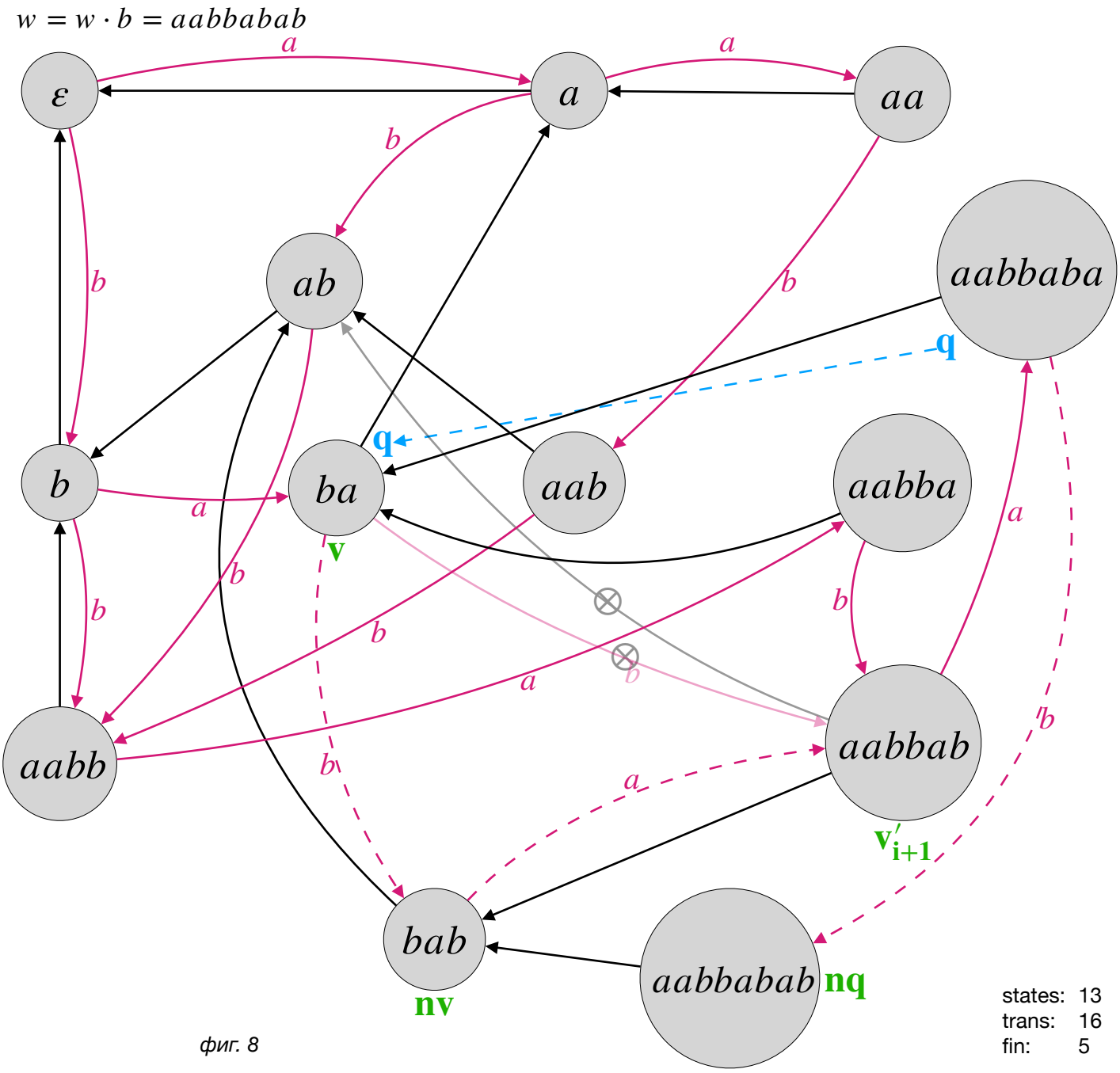
$nq = aabbab;\ q = aabba$ и $\delta(q,b) = \perp \Rightarrow$ нов преход $\delta(q,b) = nq;\ q = p(q) = a;\ \delta(q,b)$ е дефиниран и следователно $v'_{i+1} = \delta(q,b) = aab;\ v = q$, но $v \cdot b = a \cdot b = ab \neq aab \Rightarrow$ необходимо е ново състояние $nv = v \cdot b = ab$. Първо ще актуализираме автоматните преходи: второто ново състояние nv наследява преходите на v'_{i+1}, а преходите към nv са пренасочените преходи от v отиващи до v'_{i+1} с b и всички останали представители, които са били толкова къси, че с буквата b вече отиват до новото състояние nv, а не до $v'_{i+1} = aab$ (ще ги намерим отново с

бащината функция на $\mathcal{T}_w$). Актуализирахме автоматните преходи, сега остава да актуализираме и функцията на бащите на суфиксното дрво, което е лесно: $p(bv) = p(v'_{i+1})$; $p(nq) = p(v'_{i+1}) = nv$. Подготвяме курсора за последваща конкатенация с нова буква: $q = nq$.



фиг. 7

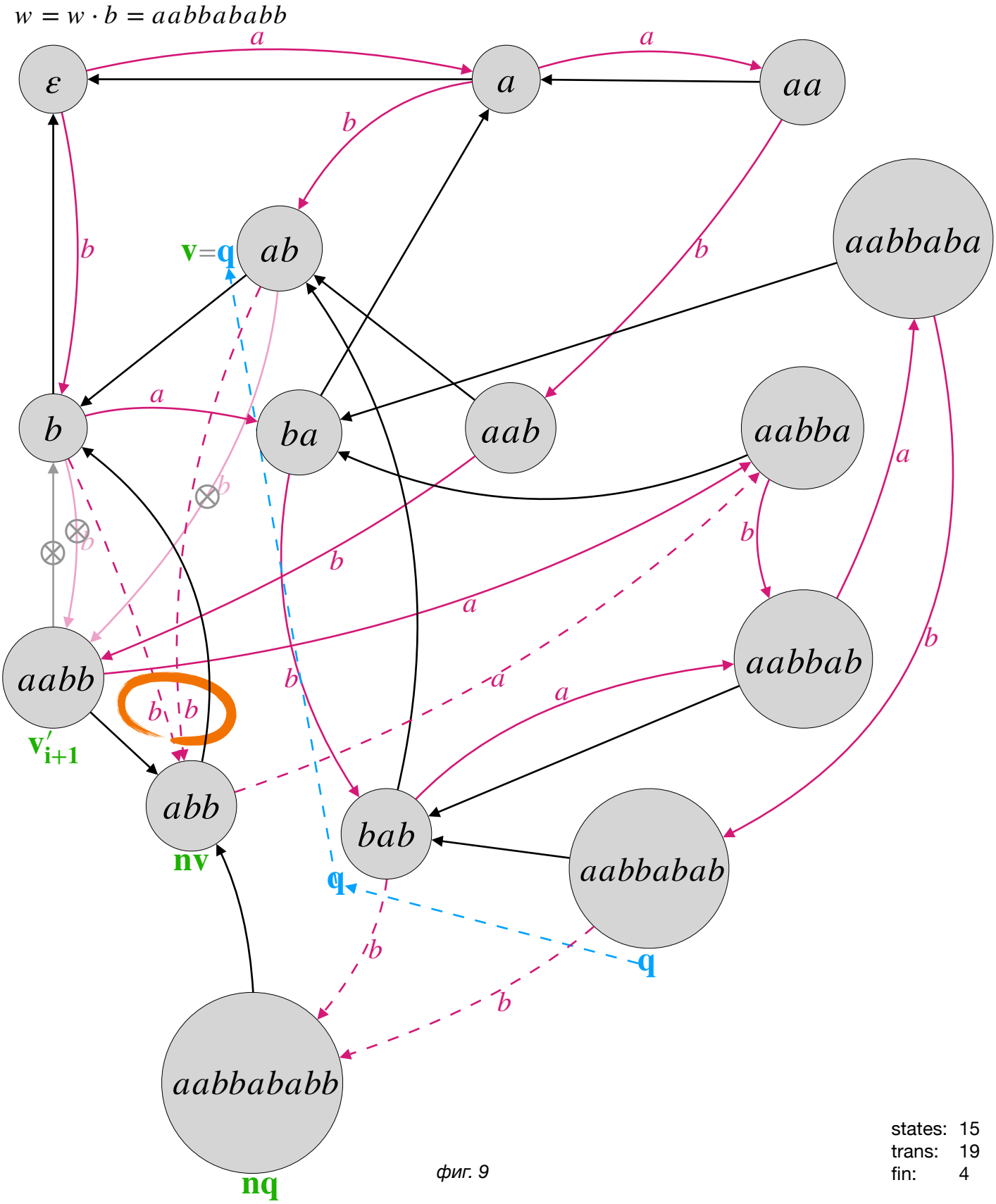
$nq = aabbaba$; $q = aabbab$ и $\delta(q,a) = \perp \Rightarrow$ нов преход $\delta(q,a) = nq$; $q = p(1) = ab$; $\delta(q,a) = \perp \Rightarrow$ нов преход $\delta(q,a) = nq$; $q = p(q) = b$; $\delta(q,a) = aabba = v'_{i+1}$; $v = q$; $v \cdot a = b \cdot a = ba \neq v'_{i+1} \Rightarrow$ необходимо е ново състояние $nv = ba$. Актуализация на автоматните преходи: nv наследява преходите на v'_{i+1}, а преходите към nv са пренасочените преходи от v отиващи до v'_{i+1} с новодобавената буква a и всички останали представители, които са били толкова къси, че с буквата a вече отиват до новото състояние nv, а не до $v'_{i+1} = aabba$ (ще ги намерим отново с бащината функция на p на суфиксното дърво $\mathcal{T}_w$ - т.е. черните насочени ребра).
Актуализирахме автоматните преходи, сега остана да се погрижим за поддръжката и на функцията на бащите, която така често ползваме. Отново, $p(nv) = p(v'_{i+1})$; $p(b) = p(v'_{i+1}) = nv$. И накрая $q = nq$.



фиг. 8

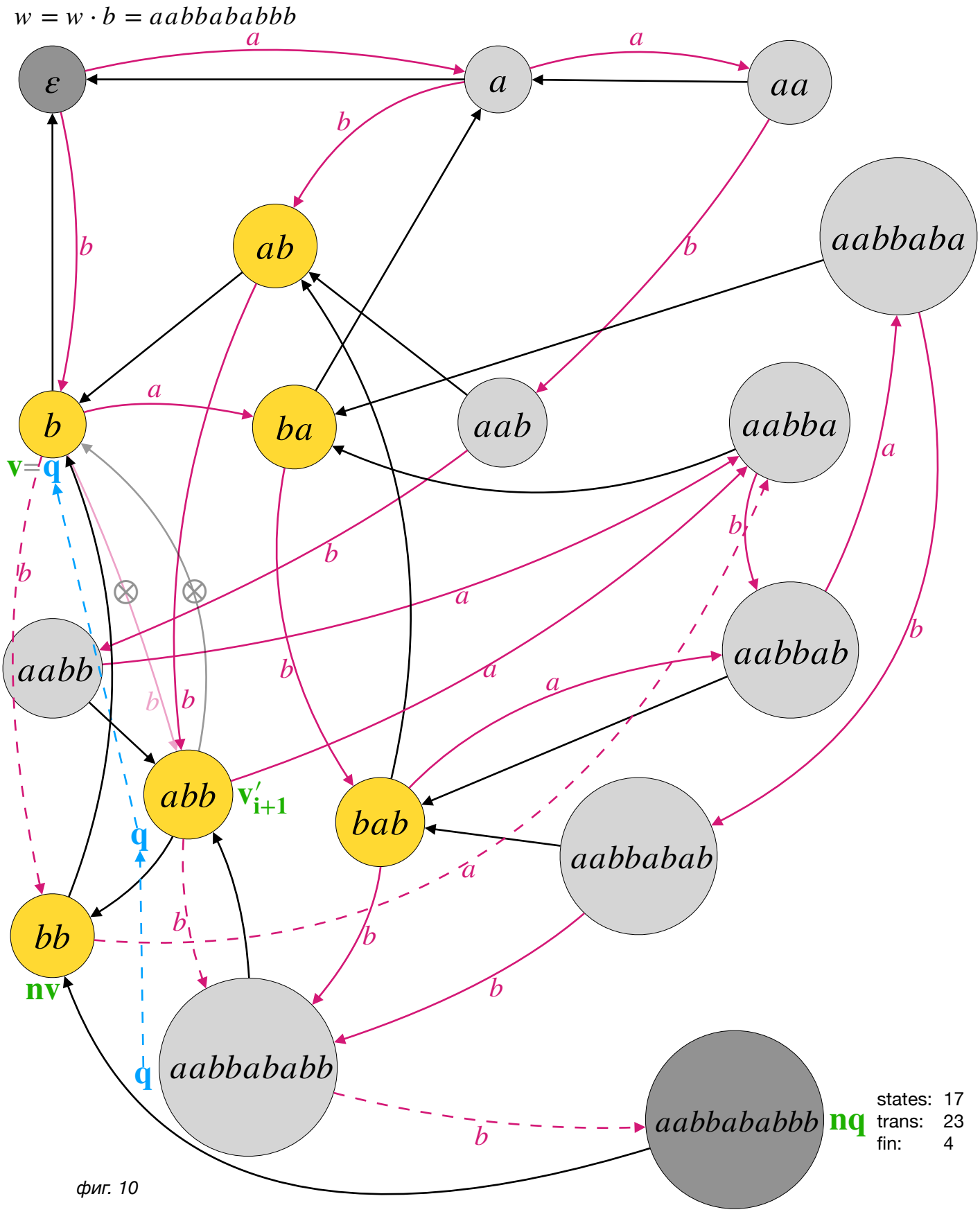
$nq = aabbabab$; $q = aabbaba$; $\delta(q,b) = \perp \Rightarrow$ нов преход $\delta(q,b) = nq$; $q = p(q) = ba$; $\delta(q,b) = aabbab = v'_{i+1}$; $v = q$; $v \cdot b = ba \cdot b = bab \neq aabbab \Rightarrow$ има нужда от ново състояние $nv = v \cdot b = bab$. Автоматни преходи: nv наследява преходите на v'_{i+1}, а преходите към nv са пренасочените преходи от v отиващи до v'_{i+1} с новодобавената буква b и всички останали представители, които са били толкова къси, че с буквата b вече отиват до новото състояние nv, а не до aab (ще ги намерим отново с бащината функция $\mathcal{T}_w$).
Актуализирахме автоматните преходи, сега остана да актуализираме функцията на бащите на суфиксното дърво. Отново, $p(nv) = p(v'_{i+1})$; $p(nq) = p(v'_{i+1}) = nv$. Накрая подготвяме курсора за добавяне на нова буква $q = nq$.

Вече съм готов да имплементирам алгоритъма, но агонията с примерите ще трябва да продължи, тъй като търся конкретен частен случай, в който трябва да се уверя, че алгоритъма няма да ме предаде (т.е. да не би случайно да съм пропуснал някаква съществена част от него по време на лекции). Това е частния случай, когато към второто ново състояние се насочват повече от едно ребро и следващата буква която се конкатенира към думата отново ще доведе до ново състояние. Опасението е, че броя на автоматните състояния може да е по-голямо отколкото би трябвало да е. Нека видим.



фиг. 9

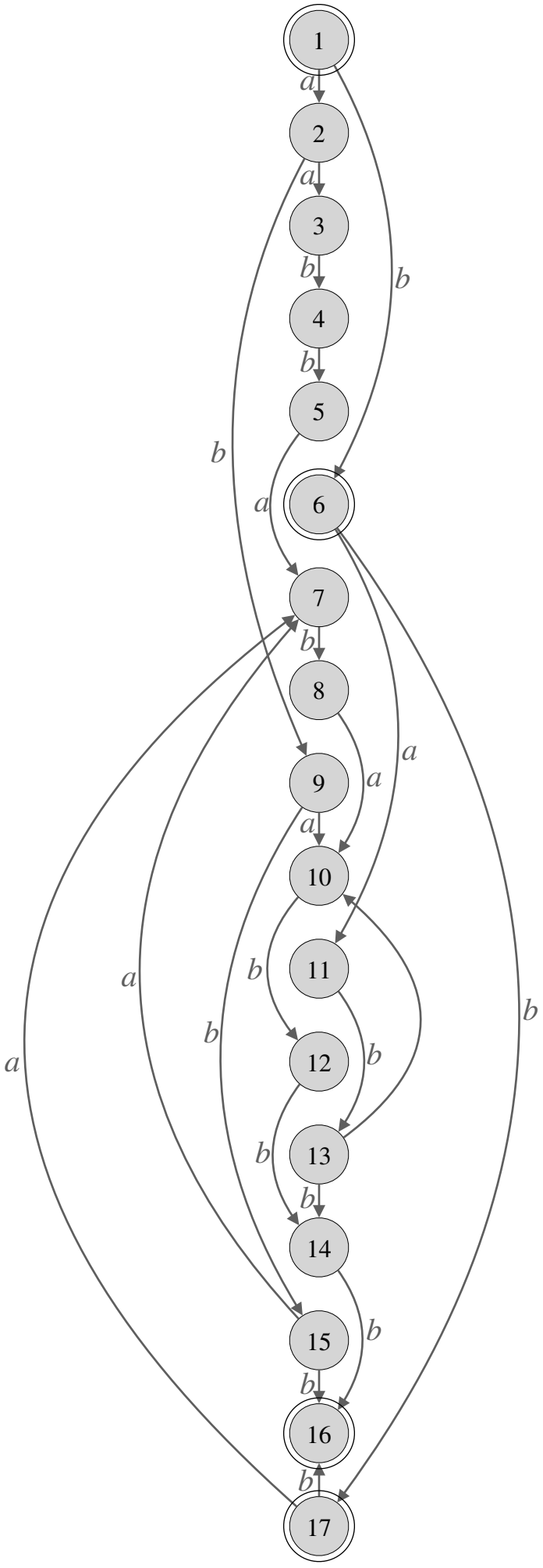
Ето го и частния случай! Сега нека конкатенираме още една буква b към думата, за да сме сигурни, че имаме и този частен случай в тестовите си сценарии.



фиг. 10

Финалните състояния както казахме в началото, може лесно да намерим след края на алгоритъма като траверсираме от най-големия суфикс - нагоре по суфиксното дърво. С тъмносив цвят се маркират финалните състояния на минималния суфиксен автомат $\mathscr{A}_w$. За целта е достатъчно да съобразим, че курсора винаги ще е в най-големия суфикс след конкатенацията на нова буква, тъй като го поддържаме готов за конкатенация със следваща буква от азбуката. Сега вече имаме достатъчно примери, с които да тестваме изхода от програмата и може да започнем имплементацията на алгоритъма на Blumer et. al.

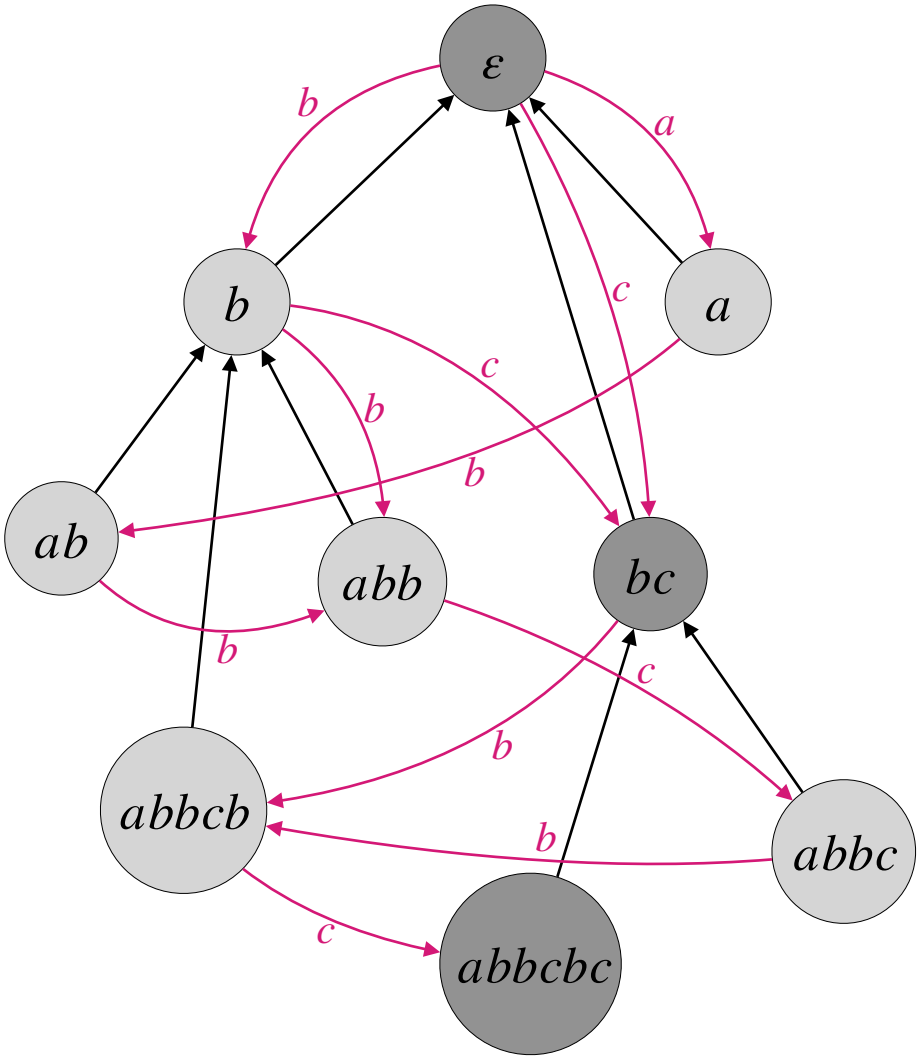
Окончателно, суфиксния автомат ще се съхрани в паметта под формата на списък на съседства и ще има следния вид:

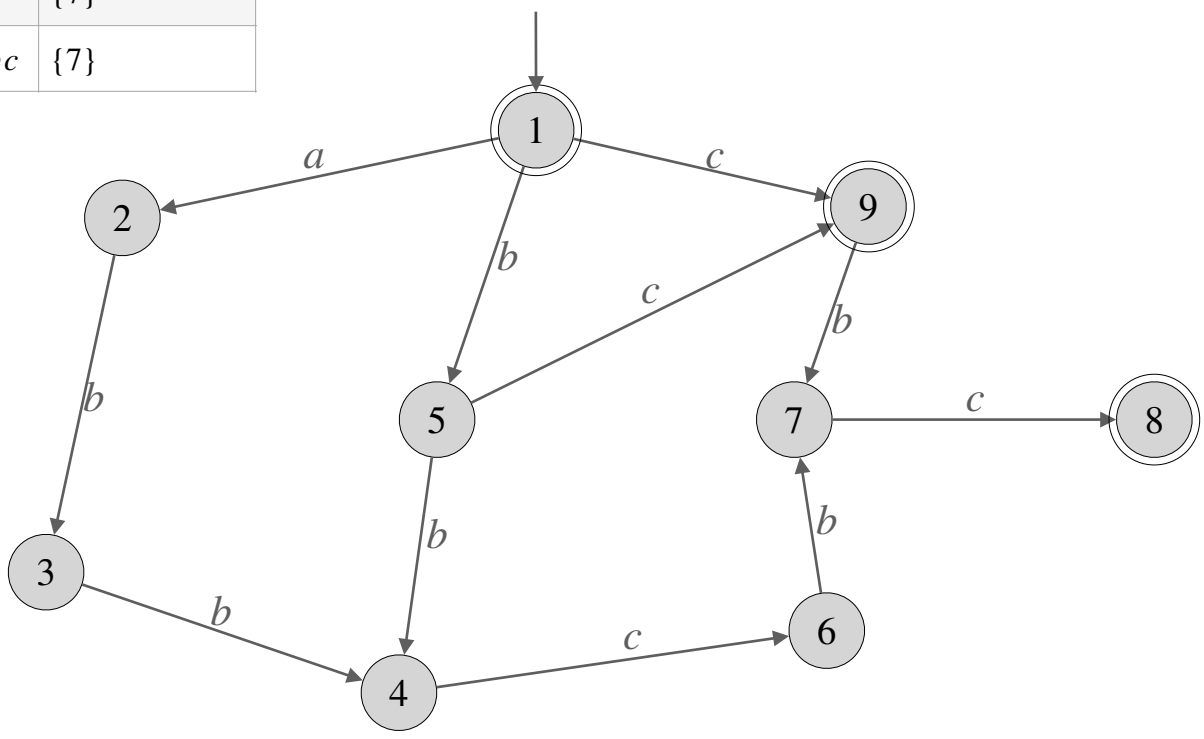


Нека за пълнота разгледаме и *off-line* алгоритъма за построяване на минимален суфиксен автомат, който не се оказва удобен за имплементация, но пък за сметка на това може да хвърли яснота за някой теоритични факти и да послужи за доказателството им. Да вземем думата $w = \underset{1}{a}\underset{2}{b}\underset{3}{b}\underset{4}{c}\underset{5}{b}\underset{6}{c}\underset{7}{c}$.

Индексацията на думата е такава, защото приемаме, че празната дума за корен и я поставяме на първата позиция с баща 0 за удобство.

$Infix_w$	$end-pos_w$
ε	{1,2,3,4,5,6,7}
a	{2}
b	{3,4,6}
c	{5,7}
ab	{3}
bb	{4}
bc	{5,7}
cb	{6}
abb	{4}
bbc	{5}
bcb	{6}
cbc	{7}
$abbc$	{4}
$bbcb$	{6}
$bcbc$	{7}
$abbcb$	{6}
$bbcbc$	{7}
$abbcbc$	{7}

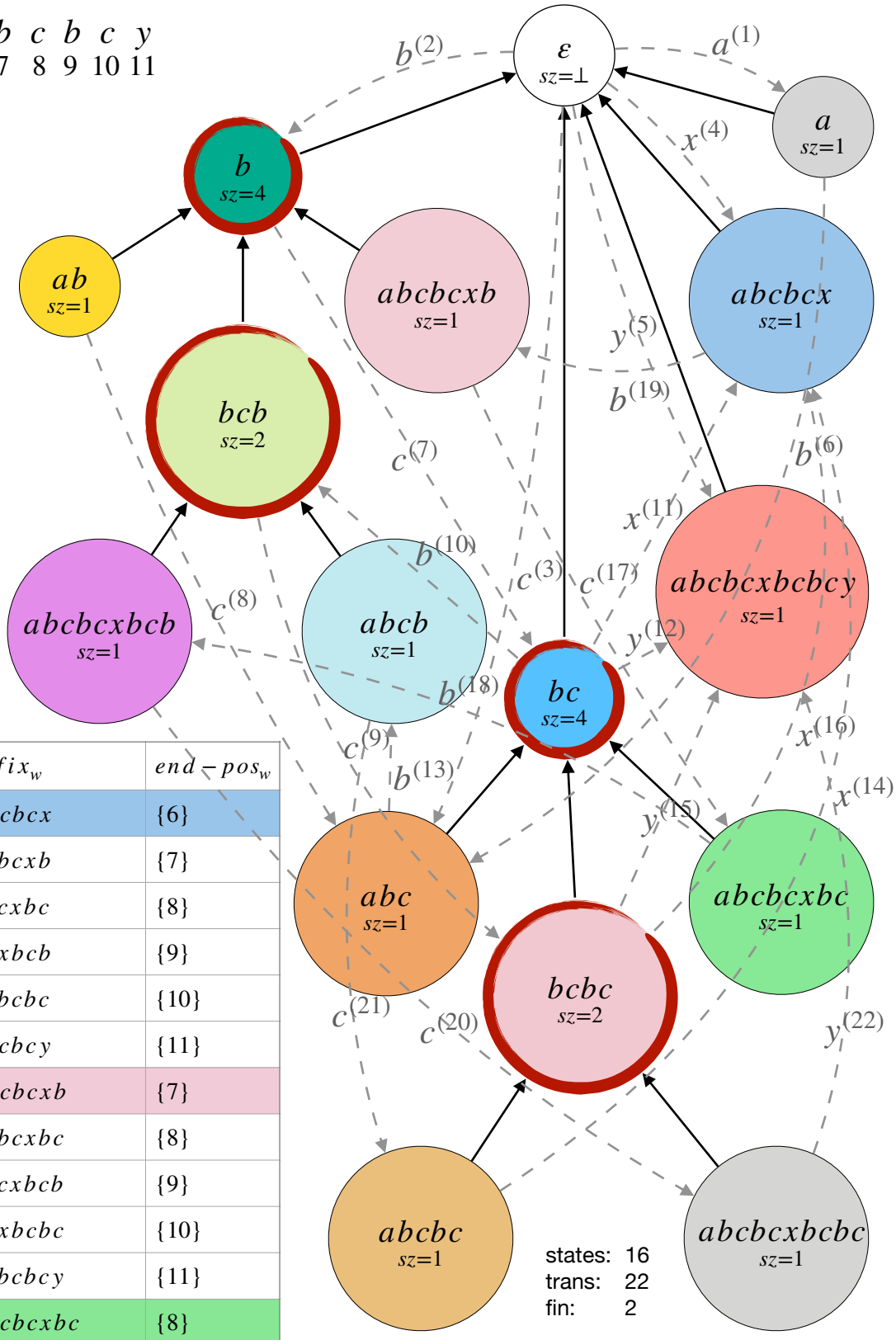




$w = a\ b\ c\ b\ c\ x\ b\ c\ b\ c\ y$
$1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10\ 11$

$Infix_w$	$end-pos_w$
ε	{0,1,2,...,11}
a	{1}
b	{2,4,7,9}
c	{3,5,8,10}
x	{6}
y	{11}
ab	{2}
bc	{3,5,8,10}
cb	{4,9}
cx	{6}
xb	{7}
cy	{11}
abc	{3}
bcb	{4,9}
cbc	{5,10}
bcx	{6}
cxb	{7}
xbc	{8}
bcy	{11}
$abcb$	{4}
$bcbc$	{5,10}
$cbcx$	{6}
$bcxb$	{7}
$cxbc$	{8}
$xbcb$	{9}
$cbcy$	{11}
$abcbc$	{5}
$bcbcx$	{6}
$cbcxb$	{7}
$bcxbc$	{8}
$cxbcb$	{9}
$xbcbc$	{10}
$bcbcy$	{11}

$Infix_w$	$end-pos_w$
$abcbcx$	{6}
$bcbcxb$	{7}
$cbcxbc$	{8}
$bcxbcb$	{9}
$cxbcbc$	{10}
$xbcbcy$	{11}
$abcbcxb$	{7}
$bcbcxbc$	{8}
$cbcxbcb$	{9}
$bcxbcbc$	{10}
$cxbcbcy$	{11}
$abcbcxbc$	{8}
$bcbcxbcb$	{9}
$cbcxbcbc$	{10}
$bcxbcbcy$	{11}
$abcbcxbcb$	{9}
$bcbcxbcbc$	{10}
$cbcxbcbcy$	{11}
$abcbcxbcbc$	{10}
$bcbcxbcbcy$	{11}
$abcbcxbcbcy$	{11}



Забележете, че големината на всеки клас на еквивалентност или иначе казано, всяко $end-pos$ множество съдържа точно толкова на брой елементи, колкото сумата от големините на множествата на децата си. Това е така, тъй като това всъщност е броя на срещания на дума прилежаща към даден клас на еквивалентност.

Брой срещания

За даден текст T, трябва да отговорим на множество заявки. За всяка поредица от символи P трябва да намерим колко пъти се среща думата P в текста T като подстринг.

Построяваме суфиксния автомат за текста T.

След което правим следната предварителна обработка: за всяко състояние v в автомата, изчисляваме броя $cnt[v]$, който е равен на размера на множеството $end-pos(v)$. Всички стрингове, които са от класа на еквивалентност представител това състояние v, се появяват в текста еднакъв брой пъти и този брой е равен на броя на позиции в това множество $end-pos(v)$, което характеризира класа на еквивалентност на върха v.

Въпреки това ние не може да построим тези множества $end-pos$ експлицитно, но пък може да пресметнем броя на всяко едно такова множество по много лесен начин.

За да го направим процедираме по следния начин. За всяко състояние, ако не е създадено от клониране (не е второ (породено) ново състояние) и не е началната празна дума ε, го инициализираме с $cnt=1$. След това ще преминем през всяко едно състояние в намаляващ ред по техния клас на еквивалентност $equivalent\_class$, и ще добавяме $cnt[v]$ на бащата на v, чрез родителската функция на суфиксното дърво, т.е. $cnt[par(v)]+=cnt[v]$. Смисъла на това е във факта, че ако стринга от класа на еквивалентност на v се появява $cnt[v]$ пъти, то тогава всички негови суфикси се появяват на същите негови завършващи позиции и следователно също се появяват $cnt[v]$ пъти.

Причината, поради която не се получава преброяване на няколко позиции няколко пъти е че добавяме позициите на състояние само до още едно състояние (родителското), следователно не може да се случи така, че едно състояние да предаде позициите си на друго състояние два пъти по два различни пътя.

Следователно може да преизчислим количествата cnt за всички състояния в автомата за $O\big(length(T)\big)$ време.

След това отговарянето на заявка е лесно. Необходимо е само да вземем стойността $cnt[t]$, където t е състоянието, отговарящо на думата от заявката, ако такова състояние съществува. В противен случай отговора е 0. Забележете, че всяка заявка ще отнема времева сложност от $O\big(length(P)\big)$.

Сега, предизвикателствата пред които сме поставени са няколко:
1. Сортиране на най-дългите представители от всеки клас на еквивалентност (всяко състояние);
2. Изчисляване на $cnt[1\ldots states]$.

Първата част не е особено тривиална. Като начало може да забележим, че в случай, при който се поражда второ ново състояние, то ще доведе до траверсиране нагоре до корена на суфиксното дърво, докато не намери състояние от което има преход със съответната буква, но този преход води до състояние с по-дълъг представител на класа си на еквивалентност и по-къс представител от първото (задължително) ново състояние за текущата итерация. Това ще доведе до случайно разбъркване в $len[1\ldots states]$ масива.

Целта ни тук е следната: за всяко състояние v да сумираме $cnt[v_j]$, където v_j, $j=1...k$ са всичките му k на брой деца и да ги прибавим към $cnt[v]$ (като предварително сме инициализирали $cnt[v_{nq}]=1$, за всяко ново състояние, което не е клонинг (не е породено), т.е. репрезентира префикс в думата w). На пръв поглед това е лесна задача. Пускаме едно обхождане в дълбочина като всеки път като излизаме от връх, актуализираме големината на класа на еквивалентност на родителя му: $cnt[v]+=cnt[child[v]]$. Но все пак, щом сме стигнали до нуждата от използване на суфиксен автомат, явно сме търсили обработка на голям обем текст и тази рекурсия би могла да препълни стека. За това най-малкото ще е необходимо да я напишем итеративно.

От друга страна, може да подходим по следния хитър начин. ще използваме counting sort, за да подредим върховете по големината на най-големия им представител в класа на еквивалентност. Това сортиране също запазва линейността на подготовката за индексация и след него лесно ще може да пресметнем $cnt[1 \dots states]$ за всяко състояние. Защо се получава така? Това е така, тъй като в суфиксното дърво всеки връх v има за баща най-големия суфикс на представителя на класа на еквивалентност на v, който е от друг клас на еквивалентност (тоест той ще е по-малък по дължина):Тук използвахме алгоритъма за сортиране чрез броене с дребната модификация, която се справя с повтарящи се числа:

```
void counting() {
      int i(0);
      for (i = 1; i <= states; ++i) ++c[len[i]];
      for (i = 1; i <= states; ++i) c[i] += c[i - 1];
      for (i = states; i; --i) a[c[len[i]]--] = i;
      for (i = states; i; --i) cnt[par[a[i]]] += cnt[a[i]];
}
```

Тук използвахме алгоритъма за сортиране чрез броене с дребната модификация, която се справя с повтарящи се числа:

COUNTING-SORT(A, B, k)

```
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

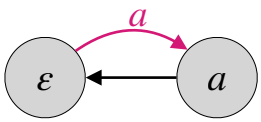
Нека все пак дадем и алтернативен вариант на имплементация за намирането на $cnt[1 \dots states]$, който също е линеен. Ето го и него:

```
void bfs(int u = 1) {
      for (int i = 1; i <= states; ++i)
            adj[par[i]].push_back(i); // adj list for suff tree
      int j(1);
      queue<int> Q;
      Q.push(u);
      int SZ = Q.size();
      while (!Q.empty()) {
            while (SZ--) {
                  int cur = Q.front();
                  Q.pop();
                  c[j++] = cur;
            for (const int child: adj[cur])
            Q.push(child);
            }
            SZ=Q.size();
      }
      for (int i = states; i ; --i)
      cnt[par[c[i]]]+=cnt[c[i]];
}
```

$w = \varepsilon$

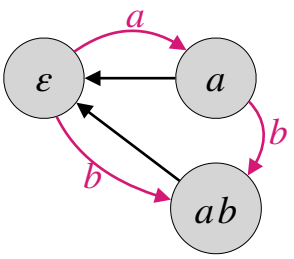
	ε	
len	0	0
	0	1

$w = w \cdot a = a$



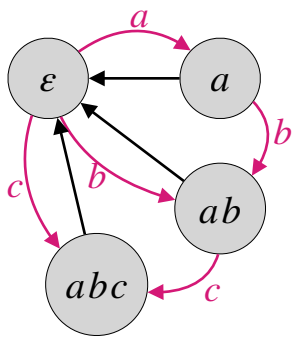
	ε	a	
len	0	0	1
	0	1	2

$w = w \cdot b = ab$



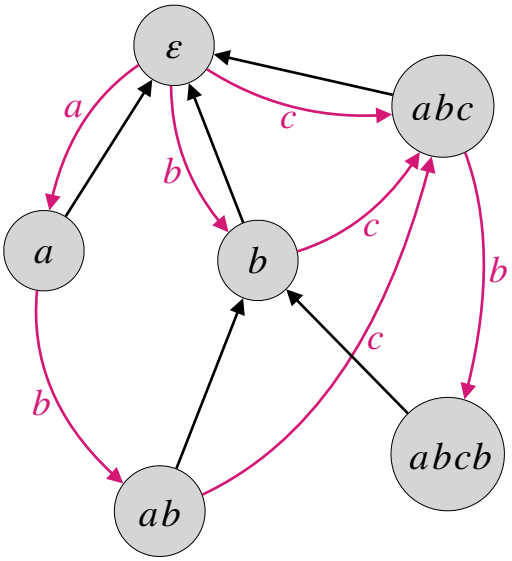
	ε	a	ab	
len	0	0	1	2
	0	1	2	3

$w = w \cdot c = abc$



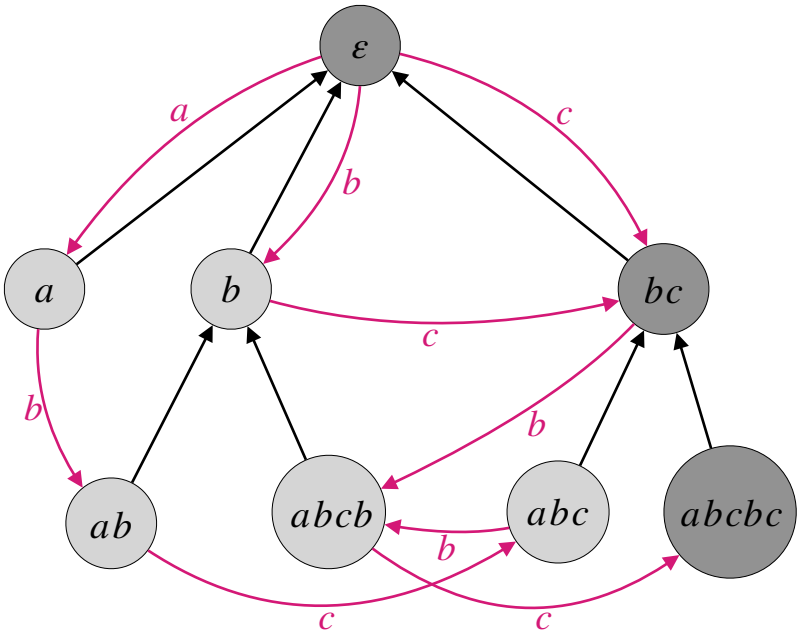
	ε	a	ab	abc	
len	0	0	1	2	3
	0	1	2	3	4

$w = w \cdot b = abcb$



	ε	a	ab	abc	abcb	b	
len	0	0	1	2	3	4	1
	0	1	2	3	4	5	6

$w = w \cdot c = abcbc$



	ε	a	ab	abc	abcb	b	abcbc	bc	
len	0	0	1	2	3	4	1	5	2
	0	1	2	3	4	5	6	7	8

След този пример вече трябва да е очевидно защо алгоритмите, макар и различни ще дават верен резултат за $cnt[1 \ldots state]$.

$counting - sort$

	ε	a	b	ab	bc	abc	abcb	abcbc	
ord	0	1	2	6	3	8	4	5	7
	0	1	2	3	4	5	6	7	8

$breadth - first - search$

	ε	a	b	bc	ab	abc	abcb	abcbc	
ord	0	1	2	6	8	3	4	5	7
	0	1	2	3	4	5	6	7	8

Първа позиция на появяване

Отново, за даден текст T трябва да отговорим на множество заявки. Този път, за всяка поредица от символи P искаме да намерим позицията на първото появяване на P в текста T (това е позицията на *началото* на първото срещане на P).

Отново построяваме суфиксен автомат за текста, но освен това, предварително трябва да изчислим и член данна $first$ за всяко състояние в автомата, т.е. за всяко състояние v искаме да намерим позицията $first[v]$ на **края** на първото му появяване. С други думи искаме да намерим предварително минималния елемент на всяко множество $end - pos$, тъй като очевидно не може да поддържаме всички множества $end - pos$ експлицитно.

За да направим това ефективно е необходимо да разширим функцията $extend$, с която конкатенираме нова буква към текста и поддържаме суфиксния автомат. Имаме най-много два случая, които трябва да разгледаме:

1. Когато добавяме ново (задължително префиксно) състояние в автомата:
   $first(nq) = len(nq) - 1;$
2. Когато клонираме връх $v = q$ до второ ново състояние vi (v'_{i+1}):
   $first(nv) = first(vi)$
   Това е така, тъй като единственото друго място, на което се появява представителя от клонираното състояние е на $first(nq)$, което очевидно е твърде далеч, защото го има и на $first(vi)$.

По този начин отговора на заявката ще е просто $first(t) - \text{length}(P) + 1$, където t е състоянието съответстващо на низа P. Отново, отговарянето на заявка ще отнема само $O\big(\text{length}(P)\big)$ времева сложност.

Всички позиции на появяване

Този път трябва да изведем всички позиции на срещания на дадена поредица от символи P в текста T.

Отново конструираме суфиксен автомат за текста T. Подобно на предишната задача, изчисляваме позицията на $first$ за всички състояния.

Очевидно $first(t)$ е част от отговора, ако t е състоянието, съответстващо на стринга за заявката P. Така взехме предвид състоянието на автомата, съдържащо P. **Какви други състояния трябва да вземем предвид?** Всички състояния, които съответстват на низове, за които P е суфикс. С други думи, трябва да намерим всички състояния, които могат да достигнат до състоянието t чрез суфиксни връзки.

Следователно, за да решим проблема, трябва да запазим за всяко състояние списък с връзки към суфикс, водещ до него. Тогава отговорът на заявката ще съдържа всички първи позиции за всяко състояние, което може да намерим при DFS/BFS обхождане, започващо от състоянието t използвайки само суфиксните връзки.

Това решение ще работи за време $O\big(\text{length}(P) + answer(P)\big)$, тъй като няма да посетим състояние два пъти (тъй като суфиксните връзки образуват дърво и няма два различни пътя водещи до едно и също състояние)

Трябва само да вземем предвид, че две различни състояния могат да имат една и съща стойност $first$. Това се случва, ако едно състояние е получено чрез клониране на друго. Това обаче не разрушава сложността, тъй като всяко състояние може да има най-много един клонинг.

Освен това може да се отървем от дублиращите се позиции, ако не изведем позициите от клонираните състояния. Всъщност, състояние което клонирано състояние може да достигне, също е достижимо от първоначалното състояние. Следователно, ако помним флаг $cloned$ за всяко състояние, може просто да игнорираме клонираните състояния и да изведем само $first$ на всички останали състояния.

Естествено има и още по-хитър начин. След като създадем списъка на съседства на суфиксното дърво, ние знаем, че клонирано състояние ще е това което има две или повече деца (без бащата), тъй като това ще индикира, че думата съответстваща на това състояние ще се намира в два различни леви контекста в w ($\exists\, a, b \in \Sigma \,|\, a \neq b \ \wedge\ a \cdot u, b \cdot u \in w$), според характеризацията на състоянията.

Лема: Нека $\mathbf{w}$ е дума, а $\mathbf{u}$ е инфикс на $\mathbf{w}$. Тогава $\mathbf{u} = \overleftarrow{\mathbf{u}}^{\mathbf{w}}$ (представител в състояние на автомата) тогава и само тогава, когато

1.) $\mathbf{u}$ е **префикс** на $\mathbf{w}$;

2.) има букви $\mathbf{a}$ и $\mathbf{b}$, които са **различни**, и за които $\mathbf{au}$ и $\mathbf{bu}$ са **инфикси** на $\mathbf{w}$.