String Matching

An Automaton Approach of Data Structure DAWGs

Shutao Shen

July 2019

Contents

1 Automata

1.1 Definition

1.1.1 Automaton

"A *(finite) automaton (with one initial state)* is given by a finite set Q, whose elements are called states, an initial state i, a subset $T \subseteq Q$ of terminal states, and a set $E \subseteq Q \times \Sigma \times Q$ of edges." [1]
Σ is used for the Alphabet Set.We use this notation for the Automaton:

$$\mathcal{A} = (Q, i, T, E)$$

1.1.2 Graph

"A *graph* $G = (V, E)$ consists of two sets: a finite set V of elements called vertices and a finite set E of elements called edges" [2] "A *directed graph* D consists of a non-empty finite set V of elements called vertices and a finite set A of ordered pairs of distinct vertices called arcs." [3] "A directed graph is acyclic if it has no directed circuit."[2] This is called *DAG(Directed acyclic graph)*. An automaton can be visualized by graph. The vertices are denoted by circles, the arcs are denoted by lines and arrows.

1.1.3 Example

The automaton in Figure 1 has: $Q = \{State_x |\ x = A, B, C, D, E, F, G, H\}$, $i = State_A$, $T = \{State_H\}$, $E = \{(State_A, a, State_B), (State_B, a, State_C),$ $(State_C, l, State_D), (State_D, s, State_E), (State_E, a, State_F), (State_F, a, State_G),$ $(State_G, l, State_H)\}$

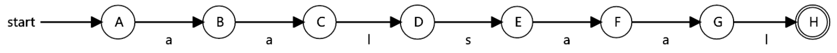


Figure 1: The non-complete DFA "aalsaal", $\Sigma = \{a, l, s\}$

1.2 Properties

1.2.1 Deterministic

"The automaton (Q, i, T, E) is *deterministic* if for each $(p, a) \in Q \times \Sigma$ there is at most one state q such that $(p, a, q) \in E$." [1] The deterministic finite automaton(DFA) of a specific language L is denoted by $\mathcal{D}(L)$.

1.2.2 Complete

"The automaton is *complete* if for each $(p, a) \in Q \times \Sigma$ there is at least one state q such that $(p, a, q) \in E$." [1]

1.2.3 Minimal

"The automaton is *minimal* if it is deterministic and if each deterministic automaton recognizing the same language maps onto it; it has the minimal number of states." [1] The minimal finite automaton of a specific language L, is denoted by $\mathcal{M}(L)$.

1.2.4 Example

Both automata in Figure 2 and Figure 3 are deterministic and minimal, the latter one has full transitions over the alphabets, so it is complete.

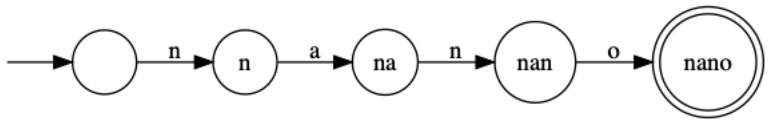


Figure 2: The minimal non-complete DFA "nano", $\Sigma = \{a, n, o\}$

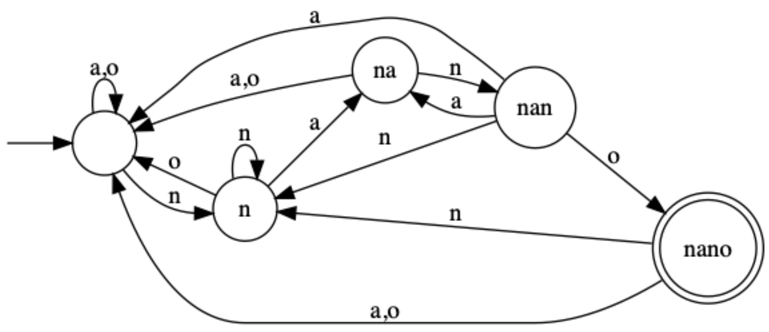


Figure 3: The minimal complete DFA "nano",$\Sigma = \{a, n, o\}$

1.2.5 Recognizable

"A word u is recognized by the automaton $\mathcal{A} = (Q, i, T, E)$ if there exists a path labeled by u from i to some state in T." [1] "A language L is recognizable if there exists an automaton $\mathcal{A}$ such that $L = Lang(\mathcal{A})$." [1] The set of all words recognized by $\mathcal{A}$ is denoted by $L = Lang(\mathcal{A})$. [1]

1.2.6 Example

The automaton recognizes "abab" and "abaaa", but not "ab" or "abaaabb". The $Lang(\mathcal{A}) = \{abab, abaaa\}$

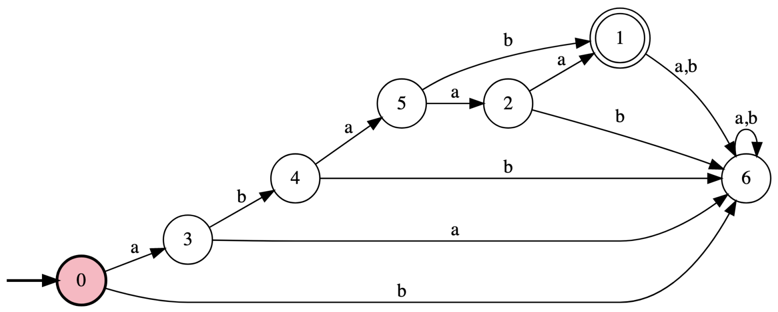


Figure 4: The complete DFA "$abab|abaaa$", $\Sigma = \{a, b\}$

1.2.7 Accept or Reject

If a word w is recognized by the automaton $\mathcal{A} = (Q, i, T, E)$, we say the automaton $\mathcal{A}$ *accepts* the word w, otherwise we say the automaton *rejects* the word.

1.2.8 Construct Automaton from RE

RE(regular expression) can be converted into NFA(nondeterministic finite automaton), we can convert the NFA to DFA using the "Workset Algorithm" and DFA to minimal DFA by the "Powerset Algorithm". [4]

2 DAWGs

Directed acyclic word graphs is one of the data structures of the suffix family. The four main members of suffix family are: *Suffix Trie*, *DAWGs*, *Suffix Tree* and *CDAWG*. The relations and differences of them can be described as a "Suffix Structures Diamond":

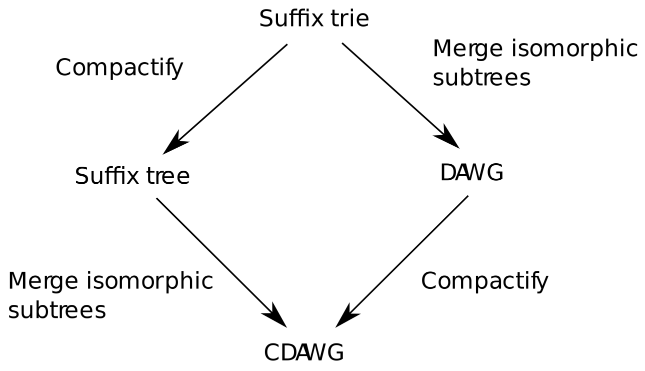


Figure 5: Suffix Structures Diamond [5]

2.1 Definition of DAWGs

“The linear size of suffix automata also called “directed acyclic word graphs” and denoted by the acronym DAWG” [1].
In the article, the above automaton approach of definition of “DAWGs” will be used, instead of the definition of stringology.
The linear size of suffix automaton(denoted by SAM in this article) and the DAWGs are essentially the same data structure. The SAM is an approach of this data structure in the theoretical computer science aspect. In comparison, the DAWGs is the same Data Structure, which always appears in the Graph Theory.
DAWG is always defined as a three elements tuple:$DAWG(w) = (V, E, L)$, where V is the “Vertices” set, E is the “Edges” set and L is the “Suffix Links” set. [6]
In comparison, the SAM can be mathematically defined as (3) in page 9, which contains the initial state and terminal states set. Since the SAM is better to know how this data structure works without other definitions such as ”Sink”, ”Explicit set”, ”Solid edges”, etc, so in this article, the SAM will be used for the analysis, and we don’t distinguish the conceptions of SAM and DAWGs and regard them as the same.

2.2 Suffix Automaton

2.2.1 Notations

Throughout the text, without additional descriptions: the first couple of lowercase english letters *(a, b,..., l)* are used for the elements of the “Alphabet” set Σ, the last few letters *(u, v, w,..., z)* are used for the words, also called texts,

strings. The length of a word w is denoted as $len(w)$, $length(w)$ or $|w|$. The empty word is denoted as ϵ or λ and has the length 0.
"The product of two words u and v, denoted by $u \cdot v$ or uv"[1], which is also called *concatenation*.
"A word v is said to be a *factor* of a word w if $w = w'vw''$ for some words $w'$and w''; it is a *prefix* of w if $w' = \epsilon$ and *suffix* if $w'' = \epsilon$" [1]
The sets of all factors, prefixes and suffixes of a word w are denoted by *Fact(w)*, *Pref(w)* and *Suff(w)*, the three sets always contain ϵ.

2.2.2 Example

a='p' is a alphabet(also called symbol)
x="hello" is a word(also called text or string)
Assume we have $x = aabb$, $y = abba$, $z = bbab$:
The *product* of x and y is $u = x \cdot y = aabbabba$
z is a *factor* of u,
x is a *prefix* of u,
y is a *suffix* of u.
$Suff(x) = \{\epsilon, b, bb, abb, aabb\}$.
$Pref(x) = \{\epsilon, a, aa, aab, aabb\}$.
$Fact(x) = \{\epsilon, a, b, aa, ab, bb, aab, abb, aabb\}$.

2.2.3 Right Occurrence

The i-th letter of the word w is denoted by $w[i]$, similarly the sub-word from i-th to j-th of the word w is denoted by $w[i..j]$, where $1 \le i < j \le |w|$.

If $u = w[i..j]$, then we say that u has a *right occurrence* in w at position j. The set of all positions(all the js) of u's right occurrences in w is denoted by $Occur_w(u)$[6], also called end positions set.[7]

2.2.4 Function end-pos

In order to use the right occurrence set without the problem of domains, now we define a function *end-pos*: $Fact(w) \rightarrow Occur_w$ by:

$$\text{for each word } u \in \text{Fact}(w),\ \text{end-pos } (u) = \text{Occur}_w(u) \tag{1}$$

2.2.5 Example

Assume $w = ba$, we have $w \in Fact(u)$, as Figure 6 shows
$Occur_{aabbabba}(ba) = \{5, 8\}$,
$Occur_{aabbabba}(c) = \phi$.
end-pos(ba) $= \{5, 8\}$,
end-pos(c) is not defined.

Note: the index of the string(word) start at 1 here.

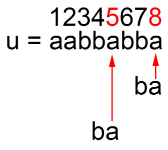


Figure 6: two occurrences of u

2.2.6 The Right End Equivalence

With the *end-pos* function, we can define "the right end equivalence relation" in the Fact(u):

$$\equiv_u^R \tag{2}$$

for $x, y \in Fact(u)$, $x \equiv_u^R y$ means x and y have the same end positions sets $Occur_u(x) = Occur_u(y)$. The equivalence class is denoted here by $[x]_u^R$ and the longest member is denoted by $(x)_u^R$. [6]
The right end equivalence is *right invariant* with respect to *concatenation* of words [6]

The right end equivalence is proved in [6] and also has the similar descriptions in [1] and [7].

2.2.7 Example

The Figure 7 shows all of the equivalence classes in the word "cccooo".

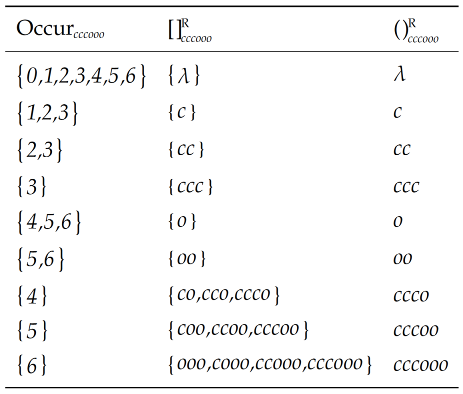
2.2.8 Mathematical Definition of Suffix Automaton

The suffix automaton of word w is defined as $\mathcal{A}(w) = (Q, i, T, E)$, where

$$\begin{aligned} &Q := \{[x]_w^R \mid x \in Fact(w)\},\\ &i := \epsilon,\\ &T \subseteq Q,\\ &E := \{([x]_w^R, a, [xa]_w^R) \mid [x]_w^R,\ [xa]_w^R \in Q \wedge a \in \Sigma\} \end{aligned} \tag{3}$$

optional, we can also define an additional set "Suffix Link":

$$L := \{([ax]_w^R, [x]_w^R) \mid [x]_w^R,\ [ax]_w^R \in Q \wedge [x]_w^R \neq [ax]_w^R \wedge a \in \Sigma\} \tag{4}$$



$Occur_{cccooo}$	$[]^R_{cccooo}$	$()^R_{cccooo}$
$\{0,1,2,3,4,5,6\}$	$\{\lambda\}$	λ
$\{1,2,3\}$	$\{c\}$	c
$\{2,3\}$	$\{cc\}$	cc
$\{3\}$	$\{ccc\}$	ccc
$\{4,5,6\}$	$\{o\}$	o
$\{5,6\}$	$\{oo\}$	oo
$\{4\}$	$\{co,cco,ccco\}$	$ccco$
$\{5\}$	$\{coo,ccoo,cccoo\}$	$cccoo$
$\{6\}$	$\{ooo,cooo,ccooo,cccooo\}$	$cccooo$

Figure 7: The equivalence class of u="cccooo" [6]

With the right invariant property, we can easily prove this automaton is the *minimal automaton*, which recognizes the language $Lang(Suff(w))$, by using the *Nerode theorem*[8].
Until now, since it has not been proved to be linear size, we call it 3-suffix-automaton. (But it is actually the same definition as DAWGs).
This automaton is in general *not* the *minimal partial DFA* of the $Lang(Suff(w))$[9], which called CDAWG.

2.3 Linear Size of Suffix Automaton

2.3.1 "a"-Right End Equivalence Relation

This part is a brief proof of the linear size property of 3-suffix-automaton.
The proof of linear size is based of the Lemma 7.2, 7.7, the Theorem 7.2 and the Corollary 7.4 of the book of M. Crochemore and C. Hancart[1]:
The Lemma 7.7 introduces a refinement of $\equiv^R_u$:

$$\equiv^R_{ua} \text{ , where } a \in \Sigma, \tag{5}$$

means the right end equivalence relation in word $(u \cdot a)$, we call it "a"-right end equivalence relation in this article.

2.3.2 Split Theorem

Theorem 7.2 can be reformulated as:
Let $w \in \Sigma^*$ and $a \in \Sigma$, let $z = longest(Suff(wa)) \wedge z \in Occur(w)$, let $z' =$

$longest(Fact(w)) \wedge z' \equiv_w^R z$, then for each $u, v \in Fact(w)$, we have:

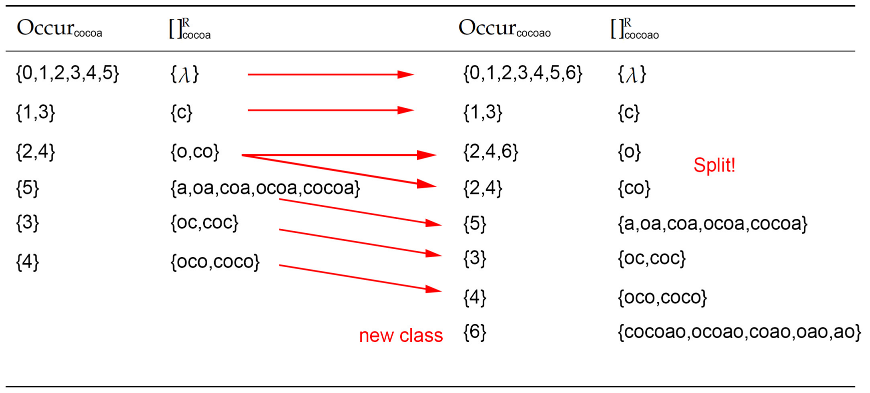
$$u \in [v]_w^R \wedge u \notin [z]_w^R \Rightarrow u \in [v]_{wa}^R$$
$$u \in [z]_w^R \Rightarrow \begin{cases} u \in [z]_{wa}^R, & if |u| \leq |z| \\ u \in [z']_{wa}^R, & \text{otherwise} \end{cases} \tag{6}$$

The first equation shows the equivalence class can be mapped to a new equivalence class when a new symbol is added. The second equation shows the $[z]_w^R$ might be split into 2 equivalence classes. This rule induces the "split procedure" of the algorithm, so we call it "Split Theorem" in this article. The 2 classes are: one class with the states whose length are not greater than $|z|$, one is opposite.
So for a word x with $3 \leq i \leq |x|$:
Due to the "Split Theorem", each symbol x[i], $3 \leq i \leq |x|$, increases *at most two equivalence classes.*
The states of the 3-suffix-automaton is linear size, precisely: $|x| + 1 \leq |Q| \leq 2|x| - 1$. [1] Corollary 7.4
The edges of the 3-suffix-automaton is linear size, precisely: $|x| \leq |E| \leq 3|x| - 4$. [1] Corollary 7.5

The linear size of the 3-suffix-automaton is proved, so from now on, we just call the in (3) defined 3-suffix-automaton "the linear size of Suffix Automaton"(denoted by "SAM") and it is also called "DAWG(s)".

2.3.3 Example

In Figure 8, w = "*cocoa*", a = 'o', z = "o" and z' = "co", the equivalence class $\{o, co\}$ is split due to the "Split Theorem" and one new class with new index is created. Others just map to new classes. The whole occurrence set increase 2 classes.



$Occur_{cocoa}$	$[]^R_{cocoa}$	$Occur_{cocoao}$	$[]^R_{cocoao}$
{0,1,2,3,4,5}	{λ}	{0,1,2,3,4,5,6}	{λ}
{1,3}	{c}	{1,3}	{c}
{2,4}	{o,co}	{2,4,6}	{o}
{5}	{a,oa,coa,ocoa,cocoa}	{2,4}	{co}
{3}	{oc,coc}	{5}	{a,oa,coa,ocoa,cocoa}
{4}	{oco,coco}	{3}	{oc,coc}
		{4}	{oco,coco}
		{6}	{cocoao,ocoao,coao,oao,ao}

Figure 8: Visualization of the "Split Theorem"

2.4 Suffix Function and Suffix Link

2.4.1 Suffix Function

It is necessary to introduce the conception "Suffix Function"[1]:

$$\begin{aligned} s_x: &\ \text{Fact }(x) \rightarrow \text{ Fact }(x) \\ s_x(v) = &\ \text{the longest } u \in Suff(v) \text{ such that } u \notin [v]_x^R \end{aligned} \tag{7}$$

It means, for a word u: $u \in Suff(x)$, u's all suffixes are not always in the same equivalence class with respect to x, the first several longest members might have the same $Occur_x$, but the short suffixes might have larger $Occur_x$ set.

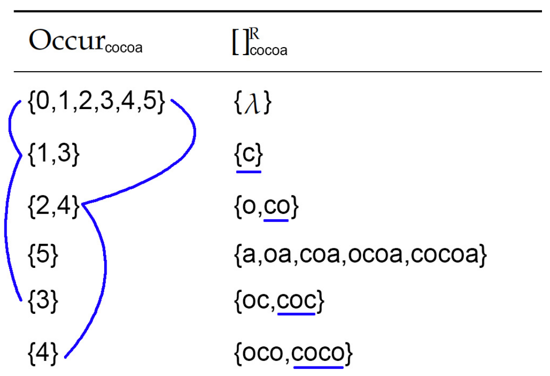
2.4.2 Suffix Link and Suffix Path

The suffix function s_x induces suffix link, for any $u \in$ state p, the link between p and the state of equivalence class $s_x(u)$ is *Suffix Link*. [1]

The suffix link describes the states with the relationship $Occur_x(u) \subsetneq Occur_x(s_x(u))$. This can be a *"suffix path"* until ϵ state, the whole path states contain the end-pos digits of $Occur_x$ set of the first set as subset.

2.4.3 Example

The suffix link of this The figure 9 shows all of the equivalence classes of word $x =$ "*cocoa*":
the suffix set of element $u =$ "*coco*" is $Suff(coco) = \{coco, oco, co, o, \epsilon\}$,
the suffix function: $s_x(coco) = co$.



$Occur_{cocoa}$	$[]^R_{cocoa}$
{0,1,2,3,4,5}	$\{\lambda\}$
{1,3}	{c}
{2,4}	{o,co}
{5}	{a,oa,coa,ocoa,cocoa}
{3}	{oc,coc}
{4}	{oco,coco}

Figure 9: Suffix link and suffix path

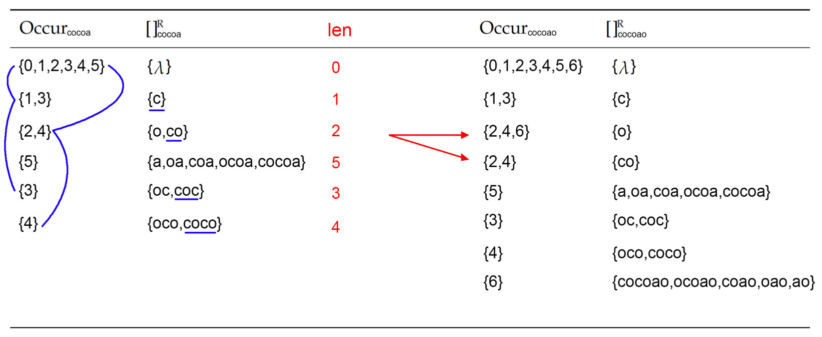
2.5 Construction of SAM: On-line Algorithm

2.5.1 Main Idea

“The algorithm processes the text from left to right. At each step, it reads the next letter of the text and updates the DAWG built so far.”[7]
The algorithm can handle the change of the text on the way of the writing, so it is called ”on-line”, and it is suitable for changing texts, such as the text editor.
The main idea of the construction is to change the equivalence classes of $\equiv_u^R$ to the equivalence classes of $\equiv_{ua}^R$, where u is the current text and a is the new input symbol.
Use the “Split Theorem”(6) in page 11 to redirect the old equivalence classes to the new equivalence classes. In the process, some states might be added into the SAM, as we know that the number of the states increases at most 2 for each symbol. Since the new symbol is added, a new end-pos number is generated, so the number of the states increases at least 1. A split process can be found in (8) in page 11.
Since we are constructing an automaton, the terminology “state” is use to replace the equivalence class.

2.5.2 Length of a state

Since the Q-set of the SAM is defined as the equivalence classes set with respect to the equivalence relation. Length of a state is defined as the longest element of the equivalence class, the same as the length of $(x)_u^R$.



Occur$_{cocoa}$	$[]^R_{cocoa}$	len	Occur$_{cocoao}$	$[]^R_{cocoao}$
{0,1,2,3,4,5}	{λ}	0	{0,1,2,3,4,5,6}	{λ}
{1,3}	{c}	1	{1,3}	{c}
{2,4}	{o,co}	2	{2,4,6}	{o}
{5}	{a,oa,coa,ocoa,cocoa}	5	{2,4}	{co}
{3}	{oc,coc}	3	{5}	{a,oa,coa,ocoa,cocoa}
{4}	{oco,coco}	4	{3}	{oc,coc}
			{4}	{oco,coco}
			{6}	{cocoao,ocoao,coao,oao,ao}

Figure 10: Length of states

The concept of state length can also be reformulated as the conceptions of “solid edge” and “non-solid edge” with respect to the on-line algorithm.
for states p and q:[7]

solid edge(p,q): length(q) = length(p) +1

non-solid edge(p,q): length(q) ≠ length(p) +1, which means the q state might be split.

2.5.3 On-line Algorithm Procedures

Adding one alphabet in the existed suffix automaton can be described as:
0. Initial the ϵ state, and set it as last state, len(last) = 0 and suff-link(last) = nil.
1. Read a new Alphabet a.
2. Create a new-last state, set the len(new-last) = len(last)+1.
3. Then we add the transition of *(last, a, new-last)* to E.
4. Traverse the suffix link of last state to go backward and try to find $\delta(p, a)$, where $p =$*suff-link*$^*(last)$. For each step, add the transition to the new-last if no transitions are found. Stop the traversing if a such transition found.
Then we get two situations:
5.1. If no states are found when the traverse ends, the "nil" will be reached. We set *suff-link*(new-last)= ϵ
5.2. If we find one $q = \delta(p, a)$, two cases need to be considered:
5.2.1. $len(q) = len(p) + 1$: *suff-link*(new-last)= q
5.2.2. $len(q) \neq len(p) + 1$: we need to split the node q
Loop(1, set $last =$*new-last*)

2.5.4 Explanations

The whole process maintains two states(last and new-last), the state lengths and suffix links. The procedures 0 to 3 are just trivial, and similar with the building process of normal automata.

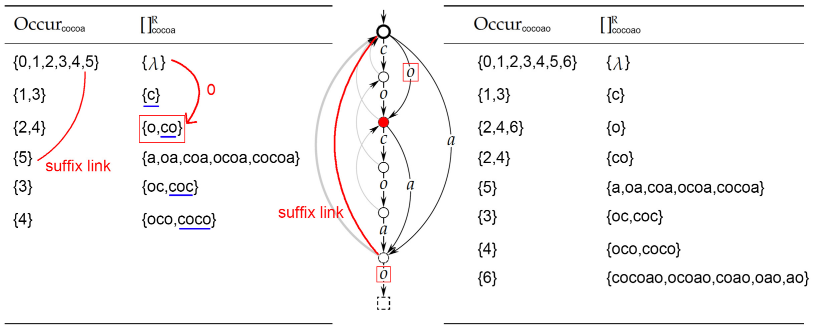
In the procedure 4, the suffix link(precisely the suffix path) of the "last" state is used to calculate the suffix link of the "new-last" state, since the automaton transfers from one state to another by consuming only one symbol each time, the first state which has the transition $\delta(p, a)$ is what we "want".

We know the suffix link links a list of states, whose elements are the suffixes of the elements of current state, but with the larger $Occur_w$ sets in the suffix path. The largest state is ϵ, so we initial the suffix link of ϵ with "nil".
For example in the Figure 10, the suffix path is: $\{4\}$ links $\{2, 4\}$ links $\{0, 1, 2, 3, 4, 5\}$, if we use the *Occur* sets to represent the states.

We know the new-last has the $Occur_{wa}(\alpha) = \{index(a)\}$, where index(a) is the length of the whole text. To calculate the suffix link of the "new-last" state is equivalent to find out which state has the second smallest set of $Occur_{wa}(\beta)$ and $index(a) \in Occur_{wa}(\beta)$, where α, β are the any elements of the state(remember this is the way we define the equivalence class). Going backward in the suffix path, $Occur_{wa}$ becomes bigger and bigger until ϵ state, so we just need to find the first one in the backwarding order. For example, in the Figure 9, which start at $\{4\}$, we find $\{2,4\}$, the second smallest set of $Occur_w$ in the suffix path.

If we want to "arrive" $State_{wa}$, firstly we need to "arrive" $State_{has\ endpos(w)}$, then we can use transition $(State_{has\ endpos(w)}, a, State_{wa})$ to finally "arrive" $State_{wa}$.

So we just traverse the suffix path of "last" state and try to find the first state which has the transition $\delta(p,a)$, marked as q, then go to step 5. We say "want", because this state q might be split due to the "Split Theorem"(6) in page 11. For example in Figure 11, state with the occurrence set of $\{6\}$ use suffix path of $\{5\}$ to calculate the suffix link, the $\{2,4\}$ is the "wanted" state.



$Occur_{cocoa}$	$[]^R_{cocoa}$
{0,1,2,3,4,5}	{λ}
{1,3}	{c}
{2,4}	{o,co}
{5}	{a,oa,coa,ocoa,cocoa}
{3}	{oc,coc}
{4}	{oco,coco}

$Occur_{cocoao}$	$[]^R_{cocoao}$
{0,1,2,3,4,5,6}	{λ}
{1,3}	{c}
{2,4,6}	{o}
{2,4}	{co}
{5}	{a,oa,coa,ocoa,cocoa}
{3}	{oc,coc}
{4}	{oco,coco}
{6}	{cocoao,ocoao,coao,oao,ao}

Figure 11: The "wanted" state

In the procedure 5, if no states are found, the suffix link of the new state should be state ϵ, because state ϵ has the occurrence set of all the indexes. If one state $q = \delta(p,a)$ is found, it must be decided whether the state should be split.

Assume that x is the longest element in state p, and y the longest in state q.
If $len(q) = len(p) + 1$, then $xa == y$, this corresponds to $z = y$ in the equivalence class q in the "Split Theorem"(6) in page 11. It means, z is the longest element in q, so the state q is not going to be split, we can directly let the suffix link of the new-last state to be state q.
If $len(q) \neq len(p) + 1$, then $xa \neq y$ but another element u in q, such that $|u| < |y| \wedge (xa = u)$, this corresponds to $z = u \wedge |z| < |y|$ in the "Split Theorem" (6) in page 11, z is not the longste, so this state is going to be split into 2 states. The split process should be easy to construct, we copy a new state to representation the equivalence class $[z]^R_{wa}$, which is similar to the algorithm in [1]:
1. create a new state q'.
2. copy all of the outgoing transitions and suff-link from state q.
3. let the $len(q') = len(p) + 1$.
4. set *suff-link(new-last)* $= q'$

5. set *suff-link(q)* $= q'$
6. redirect the link (p, a, q) to $\{(p, a, q')$, and also loop to redirect all of the suffix link state of p (using *p=Suff-link(p)*).[7]

2.5.5 Terminal Set

Terminal states are exactly the states of the suffix path of state "last".[7] So the terminal states can be gained by traversing the suffix link of the "last" state at the end of the construction.

2.5.6 Pseudo Code of On-line Algorithm

The whole process of the construction:[7]

```
SUFFIX-AUTOMATON($l$)
let $\delta$ be the transition function of (Q,i,T,E)
    $(Q, E) \leftarrow (\phi, \phi)$
    $i \leftarrow$ STATE-CREATION
    $Length[i] \leftarrow 0$
    $F[i] \leftarrow NIL$
    $last \leftarrow i$
    for $l$ from 1 up to $|x|$
        loop SA-EXTEND($l$)
    $T \leftarrow \phi$
    $p \leftarrow last$
    loop    $T \leftarrow T + \{p\}$
            $p \leftarrow F[p]$
        while $p \neq NIL$
    return $((Q, i, T, E), Length, F)$        //$F$ : Suffix Link set
```

The sub process for each step:[7]

```
SA-EXTEND($l$)
    $a \leftarrow x[l]$
    $newlast \leftarrow STATE\text{-}CREATION$
    $Length[newlast] \leftarrow Length[last] + 1$
    $p \leftarrow last$
    loop    $E \leftarrow E + \{(p, a, newlast)\}$
            $p \leftarrow F[p]$
        while $p \neq NIL$ and $\delta(p, a) = NIL$
    if $p = NIL$
        then $F[newlast] \leftarrow i$
        else $q \leftarrow \delta(p, a)$
            if $Length[q] = Length[p] + 1$
                then $F[newlast] \leftarrow q$
                //else split state
```

continue of the code:

```
                else $q' \leftarrow STATE\text{-}CREATION$
                for each letter $b$ such that $\delta(q,b) \neq NIL$
                    loop $E \leftarrow E + \{(q', b, \delta(q,b))\}$
                $Length\ [q'] \leftarrow Length\ [p] + 1$
                $F[newlast] \leftarrow q'$
                $F[q'] \leftarrow F[q]$
                $F[q] \leftarrow q'$
                loop        $E \leftarrow E - \{(p,a,q)\} + \{(p,a,q')\}$
                            $p \leftarrow F[p]$
                    while $p \neq NIL$ and $\delta(p,a) = q$
    $last \leftarrow newlast$
```

2.5.7 Complexity of On-line Algorithm

Algorithm SUFFIX-AUTOMATON computes DAWG(text) in time $\mathcal{O}(|x| \cdot log|\Sigma|)$ within $\mathcal{O}(|x|)$ space on each word x. [1]

2.5.8 Example

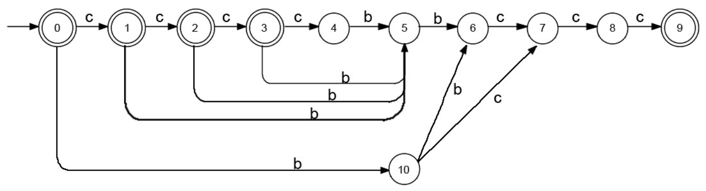


Figure 12: suffix automaton "ccccbbccc"

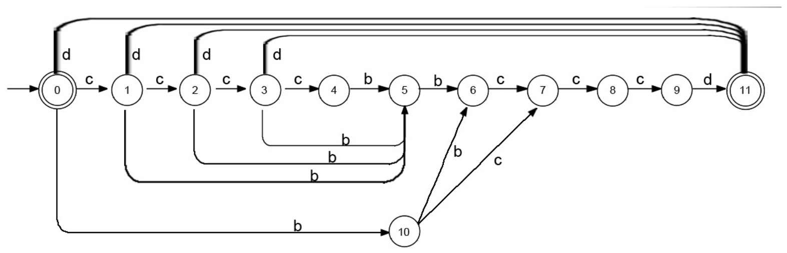


Figure 13: suffix automaton "ccccbbcccd"

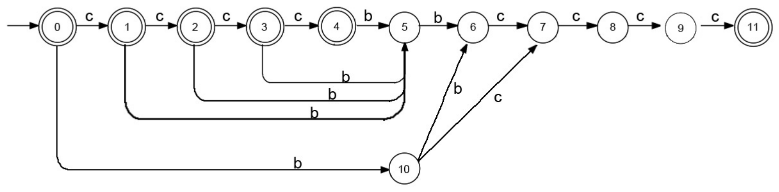


Figure 14: suffix automaton "ccccbbcccc"

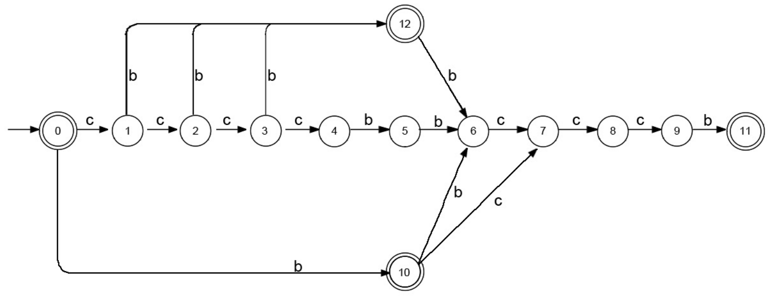


Figure 15: suffix automaton "ccccbbcccb"

3 String Matching

3.1 Definition of String Matching Problem

"Given a string P called the *pattern* and a longer string T called the *text*, the *exact matching* problem is to find all occurrences, if any, of pattern P in text T." [10]

3.2 String Matching with Automaton

The basic idea of String Matching with Automaton is quite simple. We can convert one of the strings to DFA and run the other on the automaton, if it is accepted, the string is matched.
So we have generally two solutions: 1.preprocessing of the pattern and searching the text or 2.preprocessing of the text and searching pattern

3.3 Preprocessing the pattern

3.3.1 Naive String Matching with Automaton

The whole process can be described as:
1.Build an automaton for the pattern

2.Run the *text* on the automaton
3."Accept Condition": $q \in T$
The pseudo code of the Procedure:

```
Function patternMatching(String pattern, String text):
1 DFA $\mathcal{A} = regToCompleteDFA(pattern)$;
2 State $q = i$;
3 for $k = \{0, n-1\}$:
4       if $q \in T$ then return k;
5       $q = \delta(q, text[k])$;
6 return empty;
```

The regToDFA may have $\mathcal{O}(2^m)$ states [11] and take $2^{\mathcal{O}(m)}$ time [4]

3.3.2 Lazy Automaton

Using "Lazy Automaton" can reduce the complexity to $\mathcal{O}(m+n)$.[4]

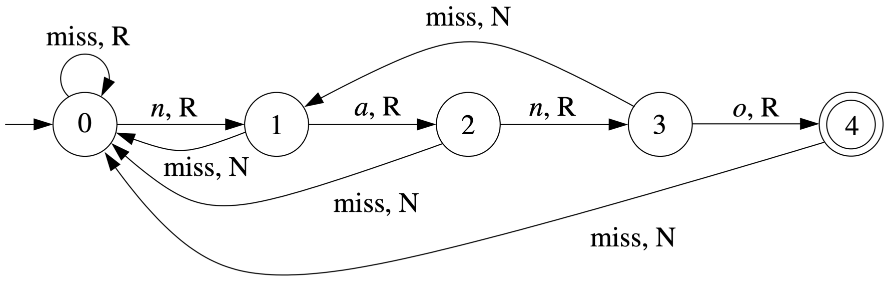


Figure 16: The pattern Lazy Automaton "nano"[4]

3.4 Preprocessing the text

3.4.1 Suffix Automaton

SAM can use to find a suffix(stop at a terminal state) or find a factor(stop at any state) of a word, and it can be built and run in linear time.

Run "ab" in this automaton, it will reach $State_2 \notin T$, so it is a factor.
Run "bb" in this automaton, it will reach $State_6 \in T$, so it is a factor.

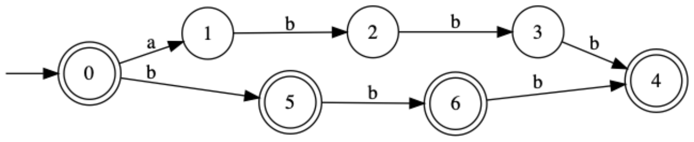


Figure 17: The suffix Automaton "abbb"

3.4.2 BDM: Backward-Dawg-Matching

The most uses of the DAWG is to use it inside an algorithm to speed up the shifting length, BDM uses $DAWG(pattern^R)$ to calculate the shift length:[7]

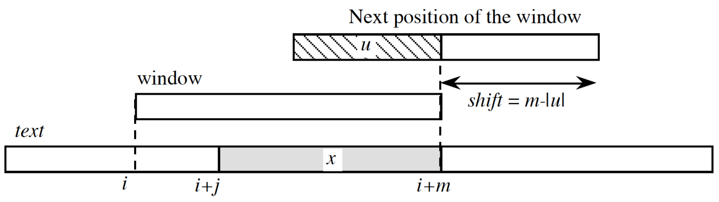


Figure 6.21. One iteration of Algorithm *BDM*.
Word *x* is the longest factor of *pat* ending at *i*+*m*.
Word *u* is the longest prefix of the pattern that is a suffix of *x*.

```
Algorithm BDM; { backward-dawg-matching algorithm }
{ use the suffix dawg DAWG(pat^R) }
begin
  i := 0;
  while i ≤ n-m do begin
    j:=m;
    while j>1 and text[i+j..i+m]∈Fac(pat) do j:=j-1;
    if j = 0 then report a match at position i;
    shift := BDM_shift[node of DAWG(pat^R)];
    i := i+shift;
  end;
end.
```

4 References

References

[1] Maxime Crochemore and Christophe Hancart. *Automata for Matching Patterns*, pages 399–462. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[2] Krishnaiyan Thulasiraman and Madisetti NS Swamy. *Graphs: theory and algorithms*. Wiley Online Library, 1992.

[3] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: theory, algorithms and applications*. Springer Science & Business Media, 2008.

[4] Javier Esparza. Automata theory, an algorithmic approach. Lecture Notes, which can be found via https://www7.in.tum.de/um/courses/auto/ws1819/autoskript.pdf, 02 2019.

[5] M. Crochemore and W. Rytter. *Jewels of Stringology: Text Algorithms*. World Scientific, 2003.

[6] Martin Senft. Suffix graphs and lossless data compression. 2013.

[7] Maxime Crochemore and Wojciech Rytter. *Text algorithms*. Maxime Crochemore, 1994.

[8] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, April 1959.

[9] A. Blumer, J. Blumer, A. Ehrenfeucht, D. Haussler, and R. McConnell. Building the minimal dfa for the set of all subwords of a word on-line in linear time. In Jan Paredaens, editor, *Automata, Languages and Programming*, pages 109–118, Berlin, Heidelberg, 1984. Springer Berlin Heidelberg.

[10] Dan Gusfield. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge university press, 1997.

[11] Gonzalo Navarro and Mathieu Raffinot. Compact dfa representation for fast regular expression search. In Gerth Stølting Brodal, Daniele Frigioni, and Alberto Marchetti-Spaccamela, editors, *Algorithm Engineering*, pages 1–13, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.