Seminar: String Matching

Directed Acyclic Word Graphs

Shutao Shen

Department of Informatics

July 12, 2019

Contents

Outline

DAWGs

Definition

“The linear size of suffix automata called Directed Acyclic Word Graphs(DAWG)” [1]

DAWGs

Suffix Structures Diamond

The differences of Suffix Trie, DAWGs, Suffix Tree and CDAWG can be described as a Suffix Structures Diamond:

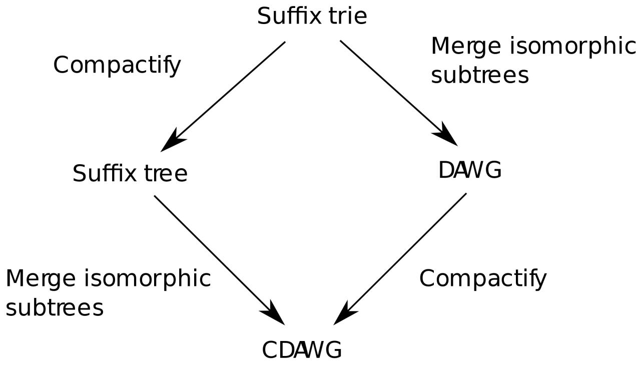


Figure: Suffix Structures Diamond [2]

Outline

Automata

Definition

Definition of (Finite) 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]

Notation

Σ is used for the Alphabet Set.

We use this notation for the automaton

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

Automata

Definition

Example

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

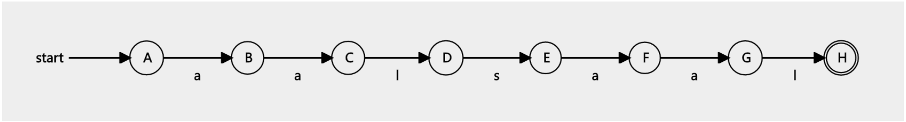


Figure: The non-complete DFA ”aalsaal”, $\Sigma = \{a, l, s\}$

Automata

Definition

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]

Notation

The set of all words recognized by $\mathcal{A}$ is denoted by $L = Lang(\mathcal{A})$. [1]

Automata

Definition

Example

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

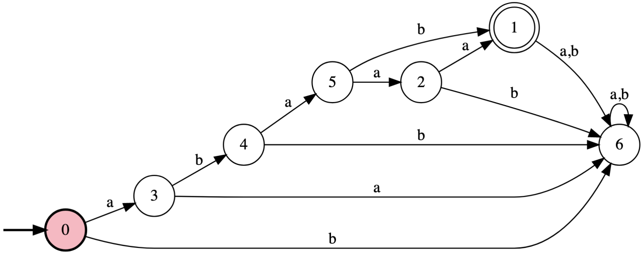


Figure: The complete DFA “$abab|abaaa$”,$\Sigma = \{a, b\}$

Automata

Properties

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]

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]

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]

Notation

For a specific language L:
deterministic finite automaton(DFA) is denoted by $\mathcal{D}(L)$.
minimal finite automaton is denoted by $\mathcal{M}(L)$.

Automata

Graph, Digraph and DAG

Graph

$G = (V, E)$

Directed Graph

pairs of distinct vertices

Directed Acyclic Graph(DAG)

“A directed graph is acyclic if it has no directed circuit.” [3]

Automata

The representation of automaton with Graph

Automaton Graph

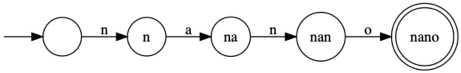


Figure: The non-complete DFA “nano”, $\Sigma = \{a, n, o\}$

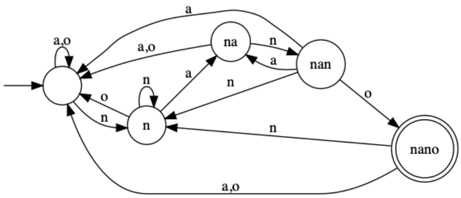


Figure: The complete DFA “nano”,$\Sigma = \{a, n, o\}$

Automata

The representation of automaton with DAG

Automaton DAG

The representation of automaton with DAG can be constructed if there are no cycles(no backward transitions), such as the following suffix automaton.

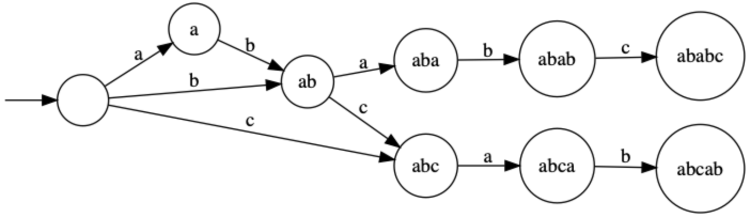


Figure: DAWG for ababc, abcab, $\Sigma = \{a, b, c\}$, $T = Q$

Automata

Run a word in automaton

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.

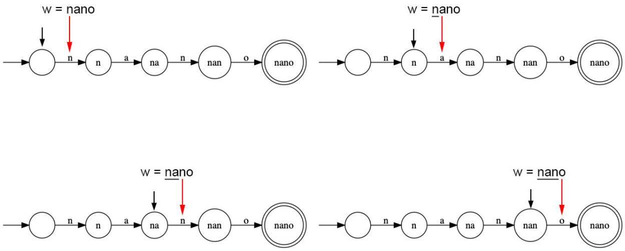


Figure: The automaton “nano” accepts “nano”

Automata

Relations between automaton and String Matching

Construct automaton from Regular Expression(RE)

RE $\rightarrow$ NFA,
NFA $\rightarrow$ DFA, ("Workset Algorithm")
DFA $\rightarrow$ minimal DFA. ("Powerset Algorithm") [4]

Relationship between String Matching and Automaton

String1 $\rightarrow$ DFA,
runDFA(String2),
if it is accepted, the string is matched.

Outline

Suffix Automaton

Definition

What is Suffix Automaton

suffix automaton of a word w:

minimal deterministic (non-necessarily complete)

recognizes the suffixes of w. [1]

Notation

The suffix automaton of word w is denoted by $\mathcal{M}(\mathit{Suff}(w))$, where $\mathit{Suff}(w)$ is the set of all suffixes of word w.

Suffix Automaton

Definition

Example

$\mathcal{M}(Suff(\text{“}abbb\text{”}))$,
recognizes all of the Suffixes of “abbb”, i.e.
$Suff(w) = \{\text{“}\epsilon\text{”}, \text{“}b\text{”}, \text{“}bb\text{”}, \text{“}bbb\text{”}, \text{“}abbb\text{”}\}$.
$T = \{State_0, State_4, State_5, State_6\}$

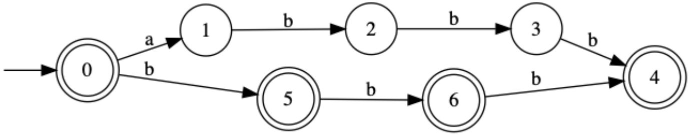


Figure: Suffix Automaton ”abbb”, $\Sigma = \{a, b\}$

Suffix Automaton

Notations

Some Notations

$\{a, b, \ldots, l\}$: Σ,

$\{u, v, w, \ldots, z\}$: words, also called texts or strings.

Examples

a='p' is a Alphabet(also called Symbol)

x="hello" is a Word(also called Text or String)

Suffix Automaton

Notations

Some Notations[1]

$u \cdot v$ or uv: *product* or *concatenation* of u and v.

$w = w'vw''$

v: *factor* of the word w

  $w' = \epsilon$: v is *prefix* of the word w

  $w'' = \epsilon$: v is *suffix* of the word w

Fact(w): The set of all factors,
Pref(w): The set of all prefixes,
Suff(w): The set of all suffixes,
those three sets always contain ϵ.

Suffix Automaton

Definition

Examples of Notations

Assume we have $x = aabb$, $y = abba$, $z = bbab$:

$$u = x \cdot y = aabbabba$$

z is a factor of u,
x is a prefix of u,
y is a suffix of u.
$Suff(x) = \{\epsilon, b, bb, abb, aabb\}$.
$Pref(x) = \{\epsilon, a, aa, aab, aabb\}$.
$Fact(x) = \{\epsilon, a, b, aa, ab, bb, aab, abb, aabb\}$.

Suffix Automaton

Right Occurrence

$Occur_w(u)$

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 $1 \leq i < j \leq |w|$, $|w|$ is the length of word 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 $Occur_w(u)$[5], also called end position set, denoted by "endpos".[6]

Suffix Automaton

Right Occurrence

Examples

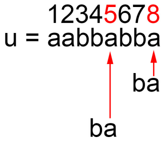


Figure: two occurrences of u

Assume $w = ba$, we have $w \in Fact(u)$, so the
$Occur_{aabbabba}(ba) = \{5, 8\}$,
$Occur_{aabbabba}(c) = \phi$.
Note: the index of the string(word) start at 1 here.

Suffix Automaton

Right Occurrence

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:

$$\text{for each word } u \in Fact(w), \textit{end-pos}(u) = Occur_w(u)$$

Note

Note that we have the same definition as [6], but not the same with [1]. The differences is that the latter endpos function map the Fact to $\mathbb{N}$, which means the leftmost occurrence.

Suffix Automaton

Right Occurrence

Examples

$$u = aabbabba$$

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

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

Suffix Automaton

The Right End Equivalence

The right end Equivalence

With the *end-pos* function, we can define “the right end equivalence relation” in the Fact(u):[5]

$$\equiv_u^R$$

for $x, y \in Fact(u)$,

$$x \equiv_u^R y \colon Occur_u(x) = Occur_u(y)$$

,

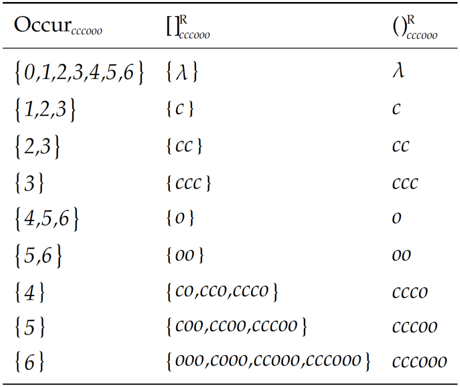
means they have the same end positions sets.
The equivalence class is denoted here by $[x]_u^R$.
The longest member is denoted by $(x)_u^R$.

The right end equivalence can be proved in [5] and also has the same descriptions in [1] and [6].

Suffix Automaton

The Right End Equivalence

The right end equivalence example



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: The equivalence class of u="cccooo" [5], (λ is ϵ)

Suffix Automaton

The Right End Equivalence

The right end Equivalence is right invariant

The right end Equivalence is *right invariant* with respect to *concatenation* of words[5]:

$$\text{Assume } a \in \Sigma, \text{ given } x \equiv_u^R y, \mathit{follows}(x \cdot a) \equiv_u^R (y \cdot a)$$

Suffix Automaton

Mathematical Definition

The Mathematical Definition of Suffix Automaton[5]

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

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

optional, we can also define an additional Set “Suffix Link”:

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

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*[7].

Suffix Automaton

Mathematical Definition

Note

Until now, since it has not been proved to be linear size, we still call it Suffix Automaton. (but it is actually the same definition as DAWGs)
The linear size will be proved later.

This automaton is in general not the *minimal partial DFA* of the $Lang(Suff(w))$[8], which called CDAWG.

Suffix Automaton

Size and Properties

Linear Size

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 introduced a refinement of $\equiv_u^R$:

$$\equiv_{ua}^R \text{, where } a \in \Sigma,$$

means the right end Equivalence relation in word $(u \cdot a)$.

Suffix Automaton

Size and Properties

Linear Size

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:

$$u \in [v]_w^R \wedge u \notin [z]_w^R \Rightarrow u \in [v]_{wa}^R.$$

$$u \in [z]_w^R \Rightarrow \begin{cases} u \in [z]_{wa}^R, & if |u| \leq |z|, \\ u \in [z']_{wa}^R, & otherwise. \end{cases} \quad (1)$$

The first shows the equivalence class can be mapped to a new equivalence class when a new symbol is added.

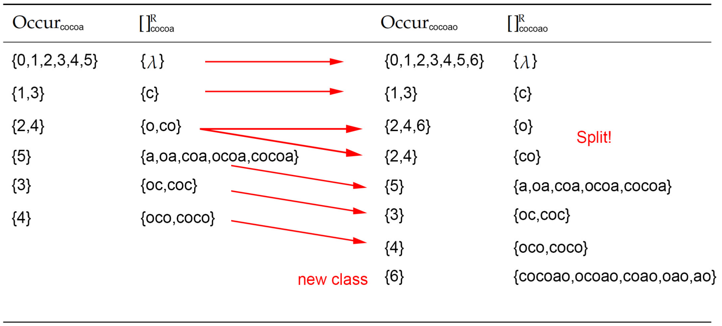
The second shows the $[z]_w^R$ might be split into 2 equivalence classes.

Suffix Automaton

Size and Properties

Example of the Theorem

$w =$ "*cocoa*", $a =$ '*o*', $z =$ "*o*" and $z' =$ "*co*", the equivalence class $\{o, co\}$ is split and one new class with new index is created. Others just map to new classes. 2 classes increase.



Occur_{cocoa}	$[\,]^R_{cocoa}$	Occur_{cocoao}	$[\,]^R_{cocoao}$
{0,1,2,3,4,5}	$\{\lambda\}$	{0,1,2,3,4,5,6}	$\{\lambda\}$
{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: Visualization

Suffix Automaton

Size and Properties

Linear Size

So due to theorem 7.2, each symbol x[i], $3 \leq i \leq |x|$, increases at most two equivalence classes. So the states of the SAM is linear, and $|x| + 1 \leq |Q| \leq 2|x| - 1$. [1] Corollary 7.4

Suffix Automaton

Size and Properties

Suffix Function

$$s_x : Fact(x) \to Fact(x)$$
$$s_x(v) = \text{the longest } u \in Suff(v) \text{ such that } u \notin [v]_x^R$$[1]

It means, if x's all suffixes are not in the same equivalence class, the first several longest might have the same $Occur_x$ but decreasing the length, the short suffixes might have bigger $Occur_x$, so they are not in the same equivalence class.

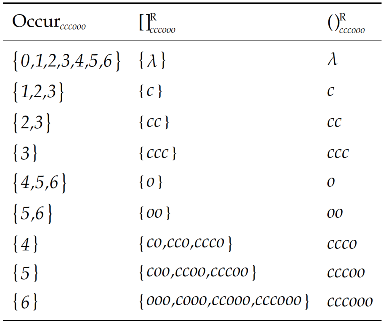
Suffix Automaton

Size and Properties

Suffix Function example

$Suff(cccooo) = \{cccooo, ccooo, cooo, ooo, oo, o, \epsilon\}$

$s_x(cccooo) = oo$



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: The equivalence class of u="cccooo" [5], (λ is ϵ)

Suffix Automaton

Size and Properties

Suffix Link

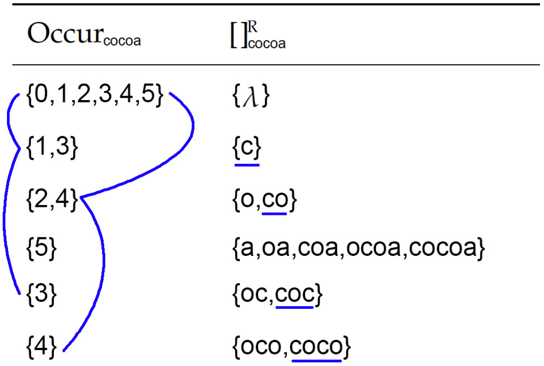
The suffix function s_x induces “suffix link”, for any u $\in$ state p, the link between p and the state of equivalence class $s_x(u)$ is Suffix Link. [1]

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

Suffix Automaton

Size and Properties

Suffix Link



$\mathrm{Occur}_{\text{cocoa}}$	$[\,]^{R}_{\text{cocoa}}$
{0,1,2,3,4,5}	$\{\lambda\}$
{1,3}	$\underline{\{\text{c}\}}$
{2,4}	{o,$\underline{\text{co}}$}
{5}	{a,oa,coa,ocoa,cocoa}
{3}	{oc,$\underline{\text{coc}}$}
{4}	{oco,$\underline{\text{coco}}$}

Figure: Suffix Link path

Suffix Automaton

DAWG and SAM

As we have already proved that the Suffix Automaton(SAM), which is defined in page 30 is linear size, so with the definition of page 4, we can say that this Suffix Automaton is a DAWG.

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. [5]

Suffix Automaton

Suffix Automaton and DAWGs

DAWG and SAM

The Suffix Automaton(SAM) 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 DAWG is an approach of the same Data Structure in Graph Theory.

Use SAM in this article

Since the SAM is better to know how this data structure works without other definitions such as Sink, Explicit set, solid edges etc. 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.

Suffix Automaton

Construction of SAM: On-line Algorithm

On-line Algorithm

“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.”[6]

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 the text editor.

Suffix Automaton

Construction of SAM: On-line Algorithm

Main Idea

Main idea:

$$\equiv_u^R \Rightarrow \equiv_{ua}^R,$$

where u is the current text and a is the new input symbol.
Use the property (1) in page 33:

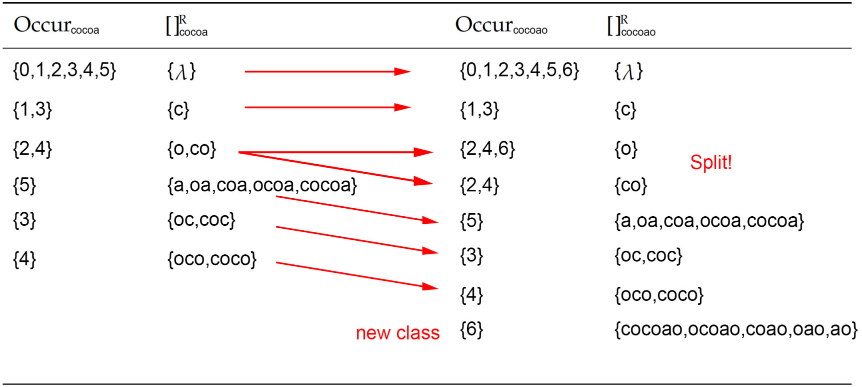
$$u \in [v]_w^R \wedge u \notin [z]_w^R \Rightarrow u \in [v]_{wa}^R.$$

$$u \in [z]_w^R \Rightarrow \begin{cases} u \in [z]_{wa}^R, & if |u| \leq |z|, \\ u \in [z']_{wa}^R, & otherwise. \end{cases} \quad (2)$$

Suffix Automaton

Construction of SAM: On-line Algorithm

Main Idea



$\mathrm{Occur}_{\text{cocoa}}$	$[\,]^{R}_{\text{cocoa}}$		$\mathrm{Occur}_{\text{cocoao}}$	$[\,]^{R}_{\text{cocoao}}$	
{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}	Split!
{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}	
		new class	{6}	{cocoao,ocoao,coao,oao,ao}	

Figure: Example of main idea

Suffix Automaton

Construction of SAM: On-line Algorithm

States need to be added

In the process, some States might be added into the SAM, as we know from (1) in page 33 that the number of the states increases at most 2 for each symbol.

And since the new symbol is added, a new end-pos number is generated, the number is the same as the index of the input symbol, the number of the states increases at least 1.

Suffix Automaton

Construction of SAM: On-line Algorithm

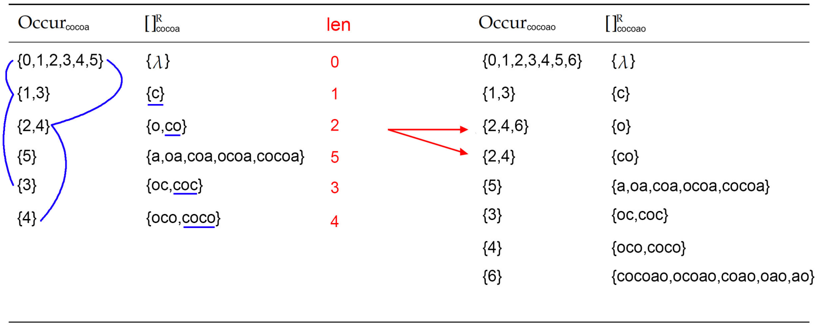
Length of the state

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

Suffix Automaton

Construction of SAM: On-line Algorithm

Example



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}	{$\underline{\text{c}}$}	1	{1,3}	{c}
{2,4}	{o,$\underline{\text{co}}$}	2	{2,4,6}	{o}
{5}	{a,oa,coa,ocoa,cocoa}	5	{2,4}	{co}
{3}	{oc,$\underline{\text{coc}}$}	3	{5}	{a,oa,coa,ocoa,cocoa}
{4}	{oco,$\underline{\text{coco}}$}	4	{3}	{oc,coc}
			{4}	{oco,coco}
			{6}	{cocoao,ocoao,coao,oao,ao}

Figure: Example of length

Suffix Automaton

Construction of SAM: On-line Algorithm

Solid Edge and Non-solid Edge

The concept of state length can also be reformulated as the conceptions of “solid edge” and “non-solid edge” in the On-line Algorithm.

So we just give descriptions of them here:
for states p and q:[6]

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

non-solid edge(p,q): length(q) $\neq$ length(p) +1,

which means the q state might be split.

Suffix Automaton

Construction of SAM: On-line Algorithm

Algorithm with the help of the properties

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 $p = \textit{suff-link}^*(\textit{last})$. Add the transition to the new-last if no Transitions found in this Iteration. Stop the traversing if found.
Then we get two situations:
5.1. If no states are found when the traverse ends, the "nil" will be reached. We set $\textit{suff-link}(\text{new-last}) = \epsilon$
5.2. If we find one $q = \delta(p, a)$, two cases need to be considered:
5.2.1. $\textit{len}(q) = \textit{len}(p) + 1$: $\textit{suff-link}(\text{new-last}) = q$
5.2.2. $\textit{len}(q) \neq \textit{len}(p) + 1$: we need to split the node q
Loop(1, set $\textit{last} = \textit{new-last}$)

Suffix Automaton

Construction of SAM: On-line Algorithm

Explanation

4. Traverse the Suffix Link of last state to go backward and try to find $\delta(p, a)$, where $p = \textit{suff-link}^*(\textit{last})$.

Since the Automaton transfer from one state to another by consuming only one symbol, we can calculate the suffix link of the “new-last” state by traversing the suffix link of the “last” state and find the first state which has the transition $\delta(p, a)$, this state is what we “want”.

why? let’s have a look.

Suffix Automaton

Construction of SAM: On-line Algorithm

Explanation

Traversing the suffix link of the “last” state and find the first state which has the transition $\delta(p, a)$, this state is what we “want”

Suffix-link: $\{4\} \to \{2, 4\} \to \{0, 1, 2, 3, 4, 5\}$,
links a list of Equivalence classes, of which elements are the suffix of the elements of current state, but with the larger $Occur_w$ in the suffix-link-paths.
The largest state is ϵ, so we initial the suff-link of ϵ with “nil”.

Suffix Automaton

Construction of SAM: On-line Algorithm

Example

In the Suffix Link path, $\{4\}$ is a subset of $\{2,4\}$ and is a subset of $\{0,1,2,3,4,5\}$.

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

Suffix Automaton

Construction of SAM: On-line Algorithm

Explanation

Traversing the suffix link of the “last” state and find the first state which has the transition $\delta(p, a)$, this state is what we “want”

State “new-last”: $Occur_{wa}(\alpha) = \{index(a)\}$
Suff-link(“new-last” state): find the second smallest set of $Occur_{wa}(\beta) \wedge (index(a) \in Occur_{wa}(\beta))$

Going backward in the suffix-link-paths $\rightarrow$ $Occur_{wa}$ becomes bigger and bigger until (ϵ state), so we just need to find the first one in the backwarding order.

Note: where α, β is any element of the state(remember this is the way we define the equivalence class).

Suffix Automaton

Construction of SAM: On-line Algorithm

Example

In the Suffix Link path, which start at $\{4\}$,
we find $\{2, 4\}$, the second smallest set of $Occur_w$ is the Suffix Link.

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

Suffix Automaton

Construction of SAM: On-line Algorithm

Explanation

Traversing the suffix link of the “last” state and find the first state which has the transition $\delta(p, a)$, this state is what we “want”

We know: suff-link (“last” state)

How to “arrive” $State_{wa}$:
firstly “arrive” $State_{has\ endpos(w)}$, then $State_{has\ endpos(w)} \rightarrow^{a} State_{wa}$.

So we just traverse the Suffix Link of ”last” state and try to find the first state which has the transition $\delta(p, a)$, marked as q, then goto step 5.
We say “want”, because this state might be split due to (1) in page 33.

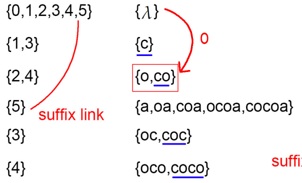
Suffix Automaton

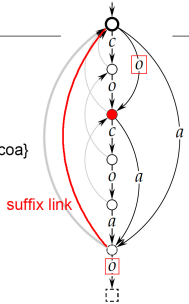
Construction of SAM: On-line Algorithm

Example

{6} use Suffix Link path of {5} to calculate the Suffix Link.

$\text{Occur}_{\text{cocoa}}$	$[\,]^R_{\text{cocoa}}$
{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}





$\text{Occur}_{\text{cocoao}}$	$[\,]^R_{\text{cocoao}}$
{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}

Suffix Automaton

Construction of SAM: On-line Algorithm

Explanation

5.1. If no states are found when the traverse ends, the “nil” will be reached. We set *suff-link*(new-last)$= \epsilon$

We know the ϵ state has the Occur set of all positions, so no smaller Occur sets are found, the suffix link must be the ϵ state.

Suffix Automaton

Construction of SAM: On-line Algorithm

Explanation

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)$= q$

In this case, assume 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 (1) in page 33.

z is the longest 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.

Suffix Automaton

Construction of SAM: On-line Algorithm

Explanation

5.2. If we find one $q = \delta(p, a)$, two cases need to be considered:
5.2.2. $len(q) \neq len(p) + 1$: we need to split the node q.

In this case, assume x is the longest element in state p, and y the longest in state q. If $len(q) \neq len(p) + 1$ then $xa \neq y$ but another element in q $|u| < |y| \wedge (xa = u)$, this corresponds to $z = u \wedge |z| < |y|$ in (1) in page 33

z is not the longste, so this state is going to be split into 2 states.

Suffix Automaton

Construction of SAM: On-line Algorithm

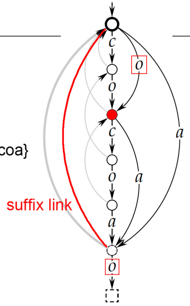
Example

$len(\{2,4\}) = 2$, $len\{\epsilon\} = 0$, $z =$ "o", $z' =$ "co", so split.

$\mathrm{Occur}_{\text{cocoa}}$	$[\,]^{R}_{\text{cocoa}}$
{0,1,2,3,4,5}	$\{\lambda\}$
{1,3}	$\{\underline{\text{c}}\}$
{2,4}	$\boxed{\{\text{o},\underline{\text{co}}\}}$
{5}	{a,oa,coa,ocoa,cocoa}
{3}	$\{\text{oc},\underline{\text{coc}}\}$
{4}	$\{\text{oco},\underline{\text{coco}}\}$

o

suffix link



$\mathrm{Occur}_{\text{cocoao}}$	$[\,]^{R}_{\text{cocoao}}$
{0,1,2,3,4,5,6}	$\{\lambda\}$
{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}

Suffix Automaton

Construction of SAM: On-line Algorithm

Explanation

5.2. If we find one $q = \delta(p, a)$, two cases need to be considered:
5.2.2. $len(q) \neq len(p) + 1$: we need to split the node q

So 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 [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)*).[6]

Suffix Automaton

Construction of SAM: On-line Algorithm

Terminal set

Terminal States are exactly the states of the suffix-link-paths of state "last". [6]

So the Terminal States can be gained by traversing the suffix link of the "last" state at the end of the construction.

Algorithm of building SAM[1]

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

Algorithm of building SAM[1]

```
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
```

Suffix Automaton

Construction of SAM: On-line Algorithm

Algorithm of building SAM[1]

```
13                  else $q' \leftarrow$ *STATE-CREATION*
14                  for each letter $b$ such that $\delta(q, b) \neq NIL$
15                      loop $E \leftarrow E + \{(q', b, \delta(q, b))\}$
16                  $Length\,[q'] \leftarrow Length\,[p] + 1$
17                  $F[newlast] \leftarrow q'$
18                  $F[q'] \leftarrow F[q]$
19                  $F[q] \leftarrow q'$
20                  loop    $E \leftarrow E - \{(p, a, q)\} + \{(p, a, q')\}$
21                          $p \leftarrow F[p]$
22                      while $p \neq NIL$ and $\delta(p, a) = q$
23        $last \leftarrow newlast$
```

Suffix Automaton

Construction of SAM: On-line Algorithm

Example

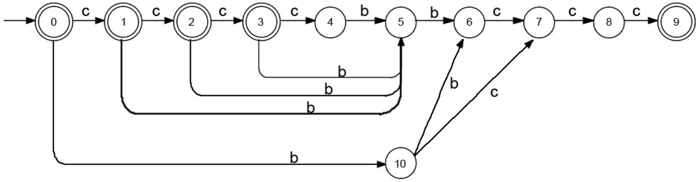


Figure: suffix automaton “ccccbbccc”

Suffix Automaton

Construction of SAM: On-line Algorithm

Example

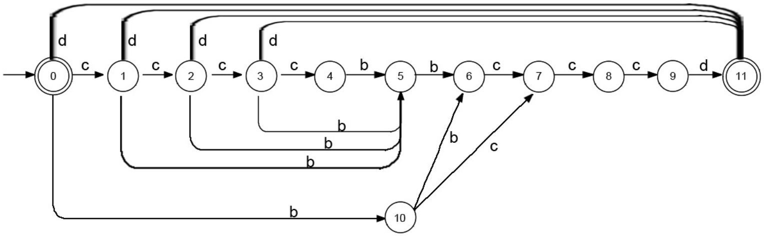


Figure: suffix automaton “ccccbbcccd”

Suffix Automaton

Construction of SAM: On-line Algorithm

Example

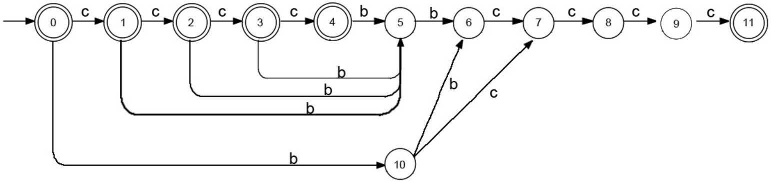


Figure: suffix automaton “ccccbbcccc”

Suffix Automaton

Construction of SAM: On-line Algorithm

Example

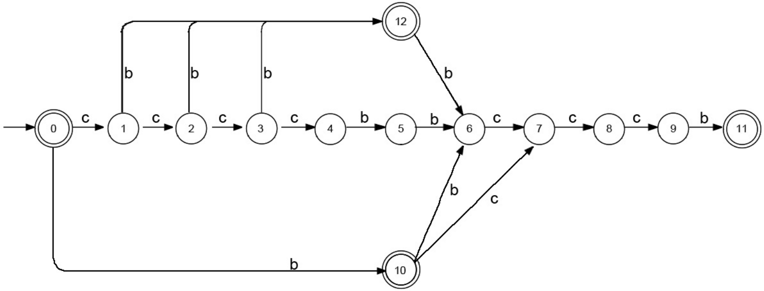


Figure: suffix automaton “ccccbbcccb”

Suffix Automaton

Construction of SAM: On-line Algorithm

Algorithm of building SAM[1]

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

Outline

String Matching

Definition of String Matching Problems

String Matching Problems

“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.” [9]

String Matching

String Matching with Automaton

Pattern Automaton

1.Preprocessing of the pattern

2.Searching text

Text Automaton

1.Preprocessing of the text

2.Searching pattern

Matching Direction(Preprocessing of the text)

1.Searching pattern prefixes from left
like: KMP, Shift Or

2.Searching pattern suffixes from right
like: BM[10]

3.Searching pattern prefixes from right
like: BDM(DAWG)[6], RF(DAWG)[11]

String Matching

Preprocessing the pattern

Naive String Matching with Automaton

1.Build a Automaton for the pattern
2.Run the *text* in the Automaton
3.Accept Condition: $q \in T$

String Matching

Preprocessing the pattern

Naive String Matching with Automaton

The pseudo code of the Procedure:

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

Notation

The regToDFA may have $\mathcal{O}(2^m)$ states [12] and take $2^{\mathcal{O}(m)}$ time [4]
Using “Lazy Automaton” can reduce the complexity to $\mathcal{O}(m+n)$.[4]

String Matching

Preprocessing the pattern

An Example of Lazy Automaton

Pattern = nano

text = nnanannanonaon

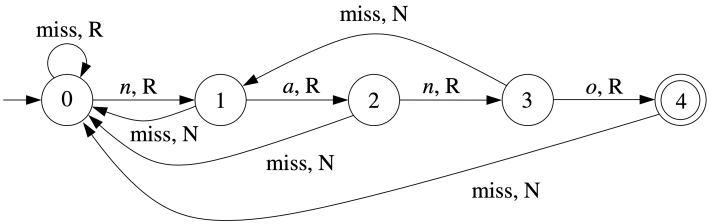


Figure: The pattern Lazy Automaton ”nano”[4]

String Matching

Preprocessing the text

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, can be built 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.

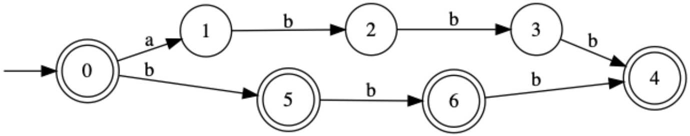


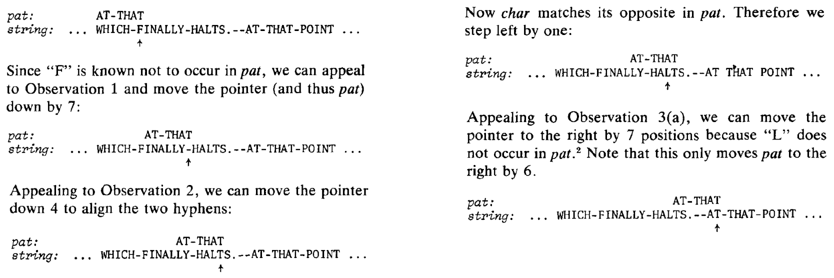
Figure: The suffix Automaton “abbb”

String Matching

Searching pattern From right to left: BM

BM[6]

“More information is gained by matching the pattern from the right than from the left” [10]:



```
pat:          AT-THAT
string:  ...  WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                    ↑
```

Since “F” is known not to occur in *pat*, we can appeal to Observation 1 and move the pointer (and thus *pat*) down by 7:

```
pat:                AT-THAT
string:  ...  WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                           ↑
```

Appealing to Observation 2, we can move the pointer down 4 to align the two hyphens:

```
pat:                     AT-THAT
string:  ...  WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                               ↑
```

Now *char* matches its opposite in *pat*. Therefore we step left by one:

```
pat:                     AT-THAT
string:  ...  WHICH-FINALLY-HALTS.--AT THAT POINT ...
                              ↑
```

Appealing to Observation 3(a), we can move the pointer to the right by 7 positions because “L” does not occur in *pat*.[2] Note that this only moves *pat* to the right by 6.

```
pat:                           AT-THAT
string:  ...  WHICH-FINALLY-HALTS.--AT-THAT-POINT ...
                                     ↑
```

String Matching

BM with SAM: BDM

BDM: Backward-Dawg-Matching[6]

Use *DAWG*(*pattern*^{R}) to calculate the shift length:[6]

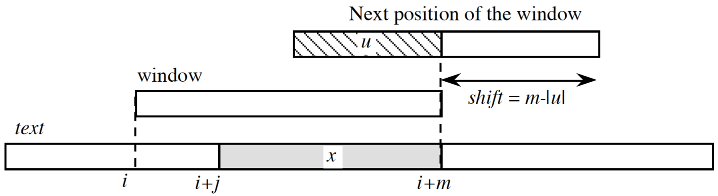


Figure 6.21. One iteration of Algorithm *BDM*.

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.

String Matching

BM with SAM: BDM

BDM: Backward-Dawg-Matching[6]

BDM:[6]

```
Algorithm BDM; { backward-dawg-matching algorithm }
{ use the suffix dawg $DAWG(pat^R)$ }
begin
  $i$ := 0;
  while $i \le n-m$ do begin
    $j$:=$m$;
    while $j>1$ and $text[i+j..i+m] \in Fac(pat)$ 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.
```

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

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

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

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.

Martin Senft.
Suffix graphs and lossless data compression.
2013.

Maxime Crochemore and Wojciech Rytter.
Text algorithms.
Maxime Crochemore, 1994.

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

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.

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

Robert S Boyer and J Strother Moore.
A fast string searching algorithm.
Communications of the ACM, 20(10):762–772, 1977.

Manindra Agrawal and P. S. Thiagarajan.
The discrete time behavior of lazy linear hybrid automata.
In Manfred Morari and Lothar Thiele, editors, *Hybrid Systems: Computation and Control*, pages 55–69, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

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.