# Generalized Aho-Corasick Algorithm for Signature Based Anti-Virus Applications

Tsern-Huei Lee
Department of Communication Engineering
National Chiao Tung University
E-Mail: tlee@banyan.cm.nctu.edu.tw

*Abstract-* **Because of its accuracy, signature matching is considered an important technique in anti-virus/worm applications. Among some famous pattern matching algorithms, the Aho-Corasick (AC) algorithm can match multiple patterns simultaneously and guarantee deterministic performance under all circumstances and thus is widely adopted in various systems, especially when worst-case performance such as wire speed requirement is a design factor. However, the AC algorithm was developed only for strings while virus/worm signatures could be specified by simple regular expressions. In this paper, we generalize the AC algorithm to systematically construct a finite state pattern matching machine which can indicate the ending position in a finite input string for the first occurrence of virus/worm signatures that are specified by strings or simple regular expressions. The regular expressions studied in this paper may contain the following operators: * (match any number of symbols), ? (match any symbol), and {*min*, *max*} (match minimum of *min*, maximum of *max* symbols), which are defined in ClamAV, a popular open source anti-virus/worm software module, for signature specification.**

## I. Introduction

Current virus/worm detection technologies can be classified into three categories, namely, protocol analysis, behavior anomaly, and pattern matching. Protocol analysis is a technique which examines the header of a packet to ensure there is no misuse of protocol fields. For example, the OID field of an SNMP packet should be a certain number of bytes. There is something wrong (say, an overflow attack) if the next expected field does not appear after this number of bytes. Behavior anomaly can be used to detect and prevent the outbreak of an attack because an infected host is likely to behave differently from a normal host. As an example, a host infected by some virus/worm may try to infect other vulnerable hosts on the Internet with port/address scanning. Therefore, one can detect an infected host with the observation of high new connection attempt rate or high failure ratio of new connection attempts [5]. Behavior anomaly can be used to detect the so-called “zero-day” attacks. However, it tends to create false positives if the normal behavior cannot be precisely specified. Finally, pattern matching is a technique of looking for specific patterns in the payload of a packet or across packets. One can utilize the strings of malicious codes contained in viruses/worms for detection. Although it is limited to known viruses/worms with identified signatures, the pattern matching technique is quite valuable because of its accuracy. Fortunately, the signature of a new virus/worm can often be quickly derived nowadays once it occurs.

The purpose of this paper is to propose construction procedures of finite state machines for signature matching. There are some well-known pattern matching algorithms such as Knuth-Morris-Pratt (KMP) [2], Boyer-Moore (BM) [3], and Aho-Corasick (AC) [4]. The KMP and BM algorithms are efficient for single pattern matching but are not scalable for multiple patterns. The AC algorithm pre-processes the patterns and builds a finite automaton which can match multiple patterns simultaneously. Another advantage of the AC algorithm is that it guarantees deterministic performance under all circumstances. As a consequence, the AC algorithm is widely adopted in various systems, especially when worst-case performance is an important design factor. Unfortunately, the AC algorithm was developed only for strings while virus/worm signatures could be specified by regular expressions. It is well known that a regular expression is equivalent to a non-deterministic finite automata (NFA) which in turn is equivalent to a deterministic finite automata (DFA). As a consequence, a straightforward approach to identify matches of a regular expression is to construct a DFA. However, the number of states in a DFA grows exponentially with the length of the regular expression in the worst case. In this paper, we present a different approach to construct a single DFA for multiple simple regular expressions.

Our constructed finite state pattern matching machine can identify the ending position of the first occurrence of virus/worm signatures which could be specified by strings and/or regular expressions. The regular expressions studied in this paper fully cover virus/worm signatures defined in ClamAV [1], an open source anti-virus software module.

The problem definition is described in Section II. In Section III, we present the construction procedure for a given set of strings together with one regular expression which contains only a single operator. The construction procedure is then generalized in Section IV for multiple regular expressions with multiple instances of operators. Finally, we draw conclusion in Section V.

## II. Problem Definition

We address in this paper the problem of constructing a finite state pattern matching machine for a set of strings *W* together with $n$ simple regular expressions $RE_1$, $RE_2$, …, and $RE_n$. The regular expression definition includes the following operators: * (match any number of symbols), ? (match any symbol), and {*min*, *max*} (match minimum of *min*, maximum of *max* symbols). We assume that every symbol is a byte. Moreover, in each $RE_k$, there is at least one *, ?, or {*min*, *max*} operator. For simplicity, we call strings in $W$ and $RE_1$, $RE_2$, …, and $RE_n$ signatures and let $\overline{W} = W \cup RE_1 \cup RE_2 \cup \ldots \cup RE_n$.

Our goal is to construct a finite state pattern matching machine that can indicate the ending position in a finite input string $x$ for the first occurrence of signature(s). A pattern matching machine is said to be valid for $\overline{W}$ if it can indicate the ending position of the first occurrence of signatures in $\overline{W}$. Our construction procedure is a generalization of the AC algorithm [4]. Throughout this paper, functions $g$, $f$, and *output* represent, respectively, the goto function, the failure function, and the output function of a finite state pattern matching machine.

We assume that a goto graph G for the set of strings $W$ has been constructed with the AC algorithm. Let $R$ denote the start state of graph G. If $W$ is an empty set, then graph G contains only state $R$ with $g(R, a) = R$ for all symbols $a$.

Some definitions are needed. We say $u$ is a prefix and $v$ is a suffix of the string $uv$. Moreover, $u$ is a proper prefix if $v$ is not empty. Likewise, $v$ is a proper suffix if $u$ is not empty. String $u$ is said to represent state $P$ in a goto graph if the shortest path from the start state to state $P$ spells out $u$. The start state is represented by the empty string. String $u$ is said to represent state $Q$ relative to state $P$ if the shortest path in the goto graph from state $P$ to state $Q$ spells out $u$.

Note that there might be a self-loop at the start state. However, it becomes a tree after removing the self-loop, if exists. In the following definitions, we ignore the self-loop. We call state $S$ the father of state $P$ if there exists a symbol $a$ such that $g(S,a) = P$. State $P$ is said to be a descendent of state $S$ if there exists a non-empty string $u$ which represents state $P$ relative to state $S$. The tree which consists of state $S$ and all its descendent states is called the sub-tree of $S$. A goto graph G is said to be "extended" with string $u$ if G is augmented with $u$ by the *enter* procedure (without the *output* function) of the AC algorithm. We say a goto graph G is extended with string $u$ from state $P$ if G is augmented with $u$ by the *enter* procedure (again, without the *output* function) using state $P$ as the start state. Extension of a goto graph with a string includes creation of new states (if necessary) and generation of the goto function. We say a string $u$ or a regular expression $RE$ is "added" to the goto graph G if a valid pattern matching machine for $W \cup \{u\}$ or $W \cup RE$ is constructed by augmenting graph G. Computation of the output function is not considered in this paper because it is the same as that in the AC algorithm. For convenience, we call any state with non-empty output function a final state.

Our constructed finite state pattern matching machine may consist of multiple separated goto graphs connected by failure functions. Scanning of an input string is equivalent to traversal of the goto graphs.

## III. One Regular Expression with a Single Operator

Let us start with the simplest case of adding one regular expression $RE_1$ with only one *, ?, or {*min*, *max*} operator to the given goto graph G.

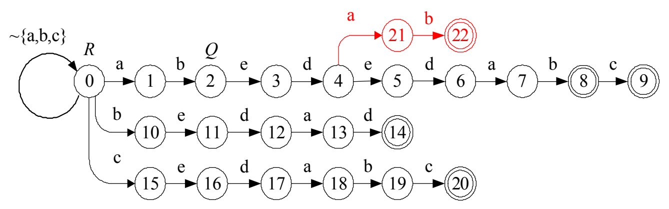
### III.A * Operator

It is clear that the * operator can be omitted for our application if it appears at the first or the last position of a regular expression. Therefore, the simplest regular expression with a single * operator is $s_1*s_2$, where $s_1$ and $s_2$ are non-empty strings. The following procedure is performed to construct a valid pattern matching machine for $\overline{W} = W \cup RE_1$.

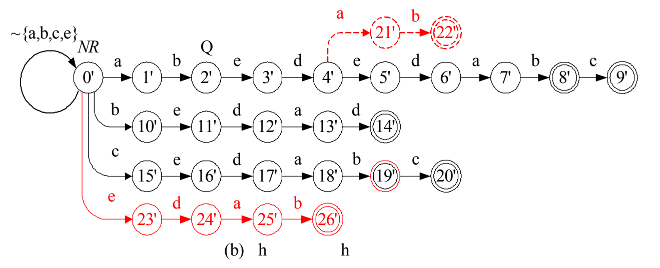
1. Duplicate the goto graph G and let $NR$ denote the start state of the duplicated graph D.
2. Extend G with string $s_1s_2$. Let $r$ denote the first symbol of $s_1$. Note that $g(R, r)$ is changed if originally $g(R, r) = R$. Denote by $Q$ the state represented by $s_1$. Let the extended graph be denoted by G'.
3. Extend the duplicated graph D with string $s_2$ and let the resulting graph be denoted by D'. Let $r$ represent the first symbol of $s_2$. Note that $g(NR, r)$ is changed if originally $g(NR, r) = NR$.
4. Extend D' with $s_1s_2$. The newly created states in this step are called virtual states. Let the extended graph be denoted by D''. It is clear that G' is contained in D'', i.e., one can find a corresponding state $S'$ in D'' for every state $S$ in G' so that the string representing $S'$ (relative to $NR$) is the same as the string representing $S$ (relative to $R$).
5. Compute independently the failure functions for graphs G' and D''. If, for some state $S'$ in D'', $f(S') = P'$ is a virtual state, then repeatedly apply $P' \leftarrow f(P')$ until $P'$ is not a virtual state and assign $f(S') = P'$.
6. For every state $Q'$ in G with representing string $us_1$, modify $f(S)$ for every state $S$ in the sub-tree of $Q'$ by assigning $f(S) = f(S')$ where $S'$ is the corresponding state in D'' of $S$ in G'. Note that $output(S)$ is updated as $output(S) \cup \{RE_1\}$ if the representing string of $S$ is $us_1vs_2$.
7. Delete the virtual states, i.e., use graphs G' and D' for traversal.

Example 1: Construct the pattern matching machine for $\overline{W} = W \cup RE_1$ where $W$ = {abededabc, bedad, cedabc} and $RE_1$ = ab*edab.

The resulting goto graph G' for $W' = W \cup$ {abedab} is shown in Figure 1(a), where ~{a, b, c} means any symbol which is not a, b, or c. All final states are shown with double circles. According to step 6, state 8 in G' is a final state because $f(8) = 26'$ which matches signature $RE_1$. Figure 1(b) illustrates the resulting goto graph D''. States 21' and 22' are virtual states. Note that although state 19 is not a final state in G', its corresponding state 19' is a final state in D'. Since we are interested in finding the first occurrence of signature(s), a goto graph can be pruned by deleting all the descendent states of a final state. For example, state 20' in D'' can be deleted without changing the match result. After pruning, a state is a final state if and only if (iff) it is a leaf state, i.e., a state without any descendent state. The failure functions for graphs G' and D'' are presented in Figures 1(c) and 1(d), respectively. According to step 3, we have $g$(0',e) = 23' although $g$(0,e) = 0.



(a) The goto graph G'.



(b) The goto graph D''.

| $i$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $f(i)$ | -- | 0 | 10′ | 11′ | 12′ | 23′ | 24′ | 25′ | 26′ | 15′ | 0 | 0 |
| $i$ | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | |
| $f(i)$ | 0 | 1 | 0 | 0 | 0 | 0 | 1 | 2 | 15 | 13′ | 2′ | |

(c) The failure function of graph G'.

| $i'$ | 0′ | 1′ | 2′ | 3′ | 4′ | 5′ | 6′ | 7′ | 8′ | 9′ | 10′ | 11′ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $f(i')$ | -- | 0′ | 10′ | 11′ | 12′ | 23′ | 24′ | 25′ | 26′ | 15′ | 0′ | 23′ |
| $i'$ | 12′ | 13′ | 14′ | 15′ | 16′ | 17′ | 18′ | 19′ | 20′ | 21′ | 22′ | 23′ |
| $f(i')$ | 24′ | 25′ | 0′ | 0′ | 23′ | 24′ | 25′ | 26′ | 15′ | 13′ | 2′ | 0′ |
| $i'$ | 24′ | 25′ | 26′ | | | | | | | | | |
| $f(i)$ | 0′ | 1′ | 2′ | | | | | | | | | |

(d) The failure function of graph D''.
Figure 1. The pattern matching machine for Example 1.

For convenience, we call state $P'$ with representing string $us$ a companion state of state $P$ with representing string $s$. As a result, every state is a companion state of itself. Moreover, if state $P_k$ is a companion state of state $P$, then either $P_k$ is $P$ or there exist states $P_1$, $P_2$, …,$P_k$, such that $f(P_i) = P_{i-1}$, $1 < i \leq k$, and $f(P_1) = P$. In step 6, we modify the failure and output functions of every state $S$ in the sub-tree of any companion state of $Q$. After the modification, we have $f(S) = P$, the state in D' such that the string $u$ representing $P$ is a proper suffix of the string $v$ representing $S$ and, if string $w$ representing any other state in D' is a proper suffix of $v$, then it is a proper suffix of $u$. This skill will be used repeatedly in this paper. For brevity, we say $P$ is the longest proper suffix state of $S$ in D'. It is worth mentioning that state $S$ in G' becomes a final state if its representing string is $us_1vs_2$ for some strings $u$ and $v$.

Traversal begins at the start state of graph G'. It stays in graph D' once it is entered. Moreover, graph D' is entered iff the failure function is consulted when a state in the sub-tree of a companion state of $Q$ is visited. We call state $Q$ a "switching state". The switching state is created because of the * operator. Since graph D' can only be entered from states in the sub-tree of a companion state of $Q$, we know that $s_1$ has already been matched if D' is entered. In fact, the signature $RE_1 = s_1{*}s_2$ is matched iff a state in G' with representing string $us_1vs_2$ is visited or a state in D' with representing string $us_2$ is visited. Therefore, the constructed pattern matching machine is valid.

## III.B ? Operator

Assume that $RE_1 = s_1?s_2$. The procedure for adding $s_1?s_2$ to G is described below.

1. Extend the goto graph G with $s_1$. Denote by $Q$ the state represented by $s_1$.
2. Extend the resulting graph from state $Q$ with $as_2$ for all symbols $a$ in $\Sigma$.
3. Determine the failure function for the resulting graph.

The basic idea of the above procedure is to extend the original goto graph G with $s_1as_2$ for all possible symbols $a$. In other words, the regular expression $s_1?s_2$ is expanded into strings $s_1as_2$ for all possible symbols $a$. It is not hard to see that the pattern matching machine constructed with the above procedure is valid.

## III.C {*min*, *max*} Operator

Assume that $RE_1 = s_1\{min, max\}s_2$. A straightforward solution of adding $RE_1$ to G is to add $s_1?^k s_2$ to G for all $k = min, \ldots, max$, where $?^k$ denotes $k$ repetitions of the ? operator. However, this solution is likely to create a huge number of new states if $max$ is a large number. A different approach which requires much fewer states is presented below.

First graph G is extended with $s_1$. Let $Q$ denote the state represented by $s_1$ and G' the resulting graph. Compute the failure function for G'. Some information has to be stored in state $Q$ to indicate that a $\{min, max\}$ operator is encountered when $Q$ is visited. The information is basically a pointer to the starting location of the remaining part of $RE_1$, i.e., $\{min, max\}s_2$. The same information is stored in all companion states of $Q$. If a companion state of $Q$ is visited, the traversal continues on graph G' and a second traversal is forked specifically to check if the remaining part of signature $RE_1$ can be matched. The goto graph T for the forked traversal is built with $\{s_2\}$. A threshold $th = max - min$ is kept and the next $min$ input symbols are skipped for the forked traversal. Assume that $s_2 = a_1a_2...a_n$. As a result, graph T consists of $n$+1 states. Number the states so that state 0 is the start state and state $i$ is the state represented by $a_1...a_i$. A counter $ctr$, with initial value 0, is maintained for the forked traversal. Update $ctr = ctr + i - f(i)$ when the failure function is consulted in state $i$. Also, $ctr$ is increased by 1 if state 0 is the current state and input symbol $a \neq a_1$. The scanning ends iff the original traversal finds a match (on graph G'), the forked traversal finds a match (on graph T), or the input string is exhausted. An additional condition for the forked traversal to end is $ctr > th$, i.e., no match is found subject to the $\{min, max\}$ constraint. Note that the original traversal may generate multiple forked traversals because the companion states of $Q$ could be visited multiple times. An example of adding regular expressions with $\{min, max\}$ operators to a goto graph G will be provided in the next section.

## IV. Multiple Instances of Operators

In this section, we consider the case of adding multiple regular expressions $RE_1$, $RE_2$, …, and $RE_n$ to a goto graph G built with a set of strings $W$. Since the ? operator can be expanded or replaced by the $\{min, max\}$ operator, we will focus on multiple instances of * and $\{min, max\}$ operators.

### IV.A Regular Expressions with * Operators Only

Assume that $RE_1$, $RE_2$, …, and $RE_n$ contain only * operators. To begin with, let us consider a simple example with only two instances of * operators $RE_1 = s_1{*}s_2{*}s_3$. For this case, the following procedure is performed to construct the pattern matching machine.

1. Create two duplicated graphs of G, called $D_1$ and $D_2$, with start states $NR_1$ and $NR_2$, respectively.
2. Extend G with $s_1s_2s_3$, $D_1$ with $s_2s_3$, and $D_2$ with $s_3$. Denote the resulting graphs by G', $D_1'$, and $D_2'$. Let $Q_1$, $Q_2$ (in graph G'), and $P$ (in graph $D_1'$) denote, respectively, the states represented by $s_1$, $s_1s_2$, and $s_2$ relative to $NR_1$.
3. Compute the failure functions independently for graphs G', $D_1'$, and $D_2'$.
4. For every state $S$ in the sub-tree of a companion state of $Q_2$ or the sub-tree of a companion state of $P$, modify $f(S) = P'$, the longest proper suffix state of $S$ in $D_2'$. The output function $output(S)$ is updated as $output(S) \cup \{RE_1\}$ if the representing string of $S$ is $us_1vs_2ws_3$. For every state $S$ in the sub-tree of a companion state of $Q_1$ but not in the sub-tree of any companion state of $Q_2$, modify $f(S) = P'$, the longest proper suffix state of $S$ in $D_1'$. The output function $output(S)$ is updated as $output(S) \cup \{RE_1\}$ if the representing string of S is $us_2vs_3$.

Note that there are two switching states $Q_1$ and $Q_2$ in G' and one switching state $P$ in $D_1'$. We say these switching states are contributed by and belong to $RE_1$. Moreover, $P$ is said to be a sibling switching state of $Q_2$ since they are created by the same * operator.

The basic idea of the above construction procedure is to distinguish three different conditions: both Condition 1 and Condition 2 are false (graph G'); Condition 1 is true and Condition 2 is false (graph $D_1'$); and both Condition 1 and Condition 2 are true (graph $D_2'$). Here Condition 1 represents the failure function is consulted in some state $S$ which is in the sub-tree of a companion state of $Q_1$ and Condition 2 means the same except that state $S$ is in the sub-tree of a companion state of $Q_2$ or the sub-tree of a companion state of $P$. Let $FQ_i$ ($i$ = 1, 2) be the flag associated with switching state $Q_i$. We set $FQ_i$ = 1 iff Condition $i$ is true. As a result, the three conditions correspond to ($FQ_1$, $FQ_2$) = (0, 0), (1, 0), and (1, 1). We call any combination of ($FQ_1$, $FQ_2$) a configuration of the pattern matching machine. It is clear that not all configurations are possibly to appear during traversal. For example, configuration (0,1) never appears because $FQ_2$ = 1 implies $FQ_1$ = 1 since $Q_2$ is in the sub-tree of $Q_1$. We say a configuration is feasible if it is possible to appear during traversal.

According to the construction procedure, each graph is extended with some suffix of $s_1s_2s_3$. The original goto graph is extended with $s_1s_2s_3$ and the output function of some states are modified so that it can be used to match strings in $W' = W \cup \{s_1s_2s_3\} \cup \{u_1s_1u_2s_2u_3s_3 \mid$ at least one of $u_1$, $u_2$, and $u_3$ is not empty and $u_1s_1u_2s_2u_3s_3$ is a prefix of some string in $W\}$. Graph $D_1$ is extended with $s_2s_3$ and the output function of some states are modified so that it can be used to match strings in $W'' = W \cup \{s_2s_3\} \cup \{us_2vs_3 \mid$ at least one of $u$ and $v$ is not empty string and $us_2vs_3$ is a prefix of some string in $W\}$. Finally, graph $D_2$ is extended with $s_3$ and therefore can be used to match strings in $W''' = W \cup \{us_3 \mid u$ is an empty string or $us_3$ is a prefix of some string in $W\}$. As a consequence, the constructed pattern matching machine is valid.

The construction procedure can be generalized to the case with an arbitrary number of * operators. Assume that there are $m$ * operators in the set of $n$ regular expressions $RE_1$, $RE_2$, …, and $RE_n$. We call the string derived from $RE_k$ by removing all the * operators $SRE_k$ (string $RE_k$). The original goto graph G is extended with $SRE_k$, $1 \leq k \leq n$, and the resulting graph is called G'. Let $Q_1$, $Q_2$, …, and $Q_m$ be the switching states in graph G'. We say two switching states $Q_i$ and $Q_j$ are identical if $u = v$ where $u$ and $v$ are the strings representing states $Q_i$ and $Q_j$, respectively. When this happens, $Q_i$ and $Q_j$ can be merged into one switching state. Assume that there are $M$ ($M \leq m$) distinct switching states, denoted by $Q_1$, $Q_2$, …, and $Q_M$, after merging identical ones. Note that a switching state may belong to multiple regular expressions because of the merging process. Denote by $FQ_i$ the flag associated with switching state $Q_i$. There are obviously $2^M$ possible combinations of ($FQ_1$, $FQ_2$, …, $FQ_M$) and each combination represents a configuration. A configuration is infeasible iff $FQ_i = 0$, $FQ_j = 1$ and $Q_j$ is in the sub-tree of $Q_i$. If there are $H$ feasible configurations (including the all-zero one), then the original goto graph G is duplicated $H$-1 times and each graph is extended with some suffix of $SRE_k$ ($1 \leq k \leq n$) so that each resulting graph corresponds to a feasible configuration.

We need to determine what suffix of $SRE_k$, $1 \leq k \leq n$, is extended in D which corresponds to feasible configuration ($FQ_1$, $FQ_2$ …, $FQ_M$). Graph D is extended with $SRE_k$ if $FQ_i = 0$ for every $Q_i$ belonging to $RE_k$. If there is at least one switching state belonging to $RE_k$ with its associated flag set, then we need to find the switching state $Q_l$ belonging to $RE_k$ which satisfies $FQ_l = 1$ and $Q_l$ is in the sub-tree of $Q_i$ for every switching state $Q_i$ belonging to $RE_k$ with $FQ_i = 1$. Once $Q_l$ is determined, graph D is extended with $SRE^{'}_k$, where $SRE^{'}_k$ is the proper suffix of $SRE_k$ which satisfies $SRE_k = uSRE^{'}_k$ and $u$ is the string representing state $Q_l$. The resulting graph after extension is denoted by D'. Note that there is a sibling switching state of $Q_i$ in graph D if $FQ_i = 0$ in the feasible configuration corresponding to graph D.

The failure functions are first computed independently for all graphs. Some modifications are necessary. Consider state $S$ in a specific graph D'. If state $S$ is in the sub-tree of a companion state of some switching state $Q$, we modify $f(S) = P$, the longest proper suffix state of $S$ in the graph corresponding to the new feasible configuration. The new feasible configuration has $FQ_i = 1$ if originally $FQ_i = 1$ or $FQ_i = 0$ and $S$ is in the sub-tree of a companion state of $Q_i$.

Traversal begins at the start state of graph G'. It switches to another goto graph corresponding to the new feasible configuration when failure occurs and the failure changes configuration. The traversal ends iff a match is found or the input string is exhausted.

### IV.B Regular Expressions with {*min*, *max*} Operators

Assume that, in addition to * operators, there are {*min*, *max*} operators contained in $RE_1$, $RE_2$, …, and $RE_n$ as well. A regular expression which contains at least one {*min*, *max*} operator is fragmented by the {*min*, *max*} operators. For example, regular expression $RE = s_1{*}s_2{*}s_3\{min_1, max_1\}s_4{*}s_5\{min_2, max_2\}s_6$ is fragmented into $re_1 = s_1{*}s_2{*}s_3$, $re_2 = s_4{*}s_5$, and $re_3 = s_6$. As in previous sections, we assume that the goto graph G built with $W$ is given. To handle $RE_1$, $RE_2$, …, and $RE_n$, we collect all regular expressions without any {*min*, *max*} operator and the first fragment of every regular expression with at least one {*min*, *max*} operator. The collection of regular expressions and first fragments are added to graph G with the procedure presented in the last section. Note that, unlike a regular expression, the first fragment $re_1$ does not make the state in any graph represented by $sre_1$ (string $re_1$) or any proper suffix of $sre_1$ a final state. Let G' denote the resulting graph.

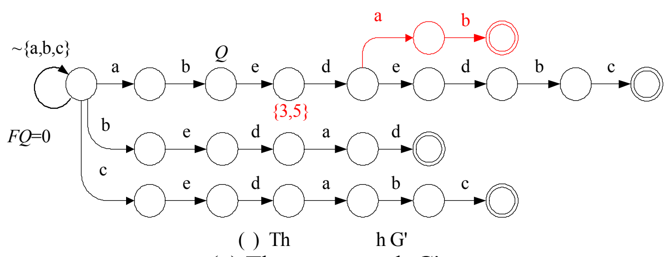
The remaining fragments of a regular expression with {*min*, *max*} operators can then be added to G' one by one. Consider the regular expression $RE = s_1{*}s_2{*}s_3\{min_1, max_1\}s_4{*}s_5\{min_2, max_2\}s_6$ as an example. The first fragment $re_1$ has been added to G to obtain G'. Since $re_1$ contains two * operations, it contributes in G' two switching states called $Q_1$ and $Q_2$. Some information is stored in the companion states of $S$ represented by $u_1s_1u_2s_2u_3s_3$ to guide a forked traversal to the starting location of the second fragment $re_2$. Similarly, for any graph $D_1$ which corresponds to a feasible configuration with $FQ_1 = FQ_2 = 0$, the same information is stored in all companion states of $S$ represented by $u_1s_1u_2s_2u_3s_3$. Consider a graph $D_2$ with corresponding configuration satisfying $FQ_1 = 1$ and $FQ_2 = 0$. The same information is stored in the companion states of $S$ represented by $us_2vs_3$. Finally, for any graph $D_3$ with corresponding configuration satisfying $FQ_1 = FQ_2 = 1$, the same information is stored in the companion states of $S$ represented by $us_3$. To summarize, the information is stored in every state which finds a match of $re_1$. In general, if the first fragment contains $K$ * operators, i.e., $re_1 = s_1{*}s_2{*}\ldots{*}s_{K+1}$, then it will contribute $K$ switching states in G', called $Q_1$, $Q_2$, …, and $Q_K$, such that $Q_i$ is in the sub-tree of $Q_{i-1}$, $1 < i \leq K$. Some information is stored in every state which finds a match of $re_1$ to guide a forked traversal to the starting location of the second fragment.

Now consider the processing of the second fragment $re_2$. The construction procedure for $re_2$ is simply to add $re_2$ to the goto graph built with an empty string. For our example, the result consists of a goto graph $T_1$ build with $\{s_4s_5\}$ which contains a switching state $Q$ represented by $s_4$ and another goto graph $T_2$ built with $\{s_5\}$. Let $P_4$ and $P_5$ be the states represented by $s_4s_5$ and $s_5$ relative to the start states of $T_1$ and $T_2$, respectively. The failure functions for graphs $T_1$ and $T_2$ are computed first independently and then modified for states in the sub-tree of $P_4$. Information is stored in states $P_4$ and $P_5$ to guide another forked traversal to the starting location of the third fragment $re_3$. For convenience, we call states $P_4$ and $P_5$ terminal states of the second fragment $re_2$. The maintained counter $ctr_1$ is increased by one if the current state is the start state of graph $T_1$ and it returns to the same state after an input symbol is processed. Assume that the failure function is consulted in state $S$. If $S$ is not in the sub-tree of $Q$, then $ctr_1$ is updated as $ctr_1 = ctr_1 + |u| - |v|$, where $u$ and $v$ are the strings representing states $S$ and $f(S)$, respectively, and $|z|$ denotes the length of string $z$. If state $S$ is in the sub-tree of $Q$, then $ctr_1$ is updated as $ctr_1 = ctr_1 + |u| - |v| - |w|$, where $u$, $v$, and $w$ are the strings representing states $S$, $Q$, and $f(S)$, respectively. In this case, the traversal continues on graph $T_2$ if the updated $ctr_1$ is smaller than or equal to $th_1 = max_1 - min_1$. In general, if the second fragment contains ($K$-1) * operators, i.e., $re_2 = s_1{*}s_2{*}\ldots{*}s_K$, then $K$ graphs, $T_1$, $T_2$, …, and $T_K$, are required such that $T_i$ is constructed with $\{s_is_{i+1}\ldots s_K\}$. Note that there are $K$-$i$ switching states, denoted by $Q_1$, $Q_2$, …, and $Q_{K-i}$, in graph $T_i$. For the traversal on graph $T_i$, $ctr_1$ is increased by one if the current state is the start state of $T_i$ and it returns to the same state after an input symbol is processed. If the failure function is consulted in state $S$ which is not in the sub-tree of any switching state, then $ctr_1$ is updated as $ctr_1 = ctr_1 + |u| - |v|$, where $u$ and $v$ are, respectively, the strings representing states $S$ and $f(S)$, and the traversal continues on graph $T_i$ if $ctr_1 \leq th_1$. If state $S$ is in the sub-tree of $Q_j$ but not in the sub-tree of $Q_{j+1}$, then $ctr_1$ is updated as $ctr_1 = ctr_1 + |u| - |v| - |w|$, where $u$, $v$, and $w$ are the strings representing states $S$, $Q_j$, and $f(S)$, respectively,. In this case, the traversal continues on graph $T_{i+j}$ if $ctr_1 \leq th_1$. Note that if the state represented by $s_i s_{i+1} \ldots s_K$ (a terminal state of $re_2$) is visited, then a forked traversal is created. The above construction procedure and traversal can be applied to any remaining fragment other than the last fragment.

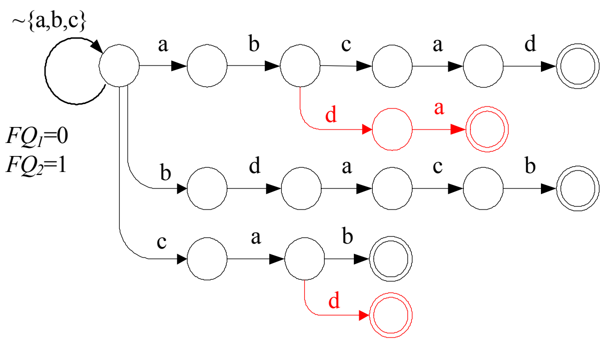
Let us consider the construction procedure for the last fragment. In our example, since $re_3$ is a string, we need only build a goto graph $T_3$ with $\{s_6\}$. Let $F$ denote the state represented by $s_6$ relative to the start state of $T_3$. The failure function for graph $T_3$ is computed independently. The signature $RE$ is matched iff state $F$ is visited. Again, if the last fragment contains ($K$-1) * operators, i.e., $re_3 = s_1{*}s_2{*}\ldots{*}s_K$, then $K$ graphs, $T_1$, $T_2$, …, and $T_K$, are required such that $T_i$ is constructed with $\{s_i s_{i+1} \ldots s_K\}$. State $F$ in graph $T_K$ represented by $s_K$ is the only final state for $RE$.

Example 2: Construct the pattern matching machine for $\overline{W} = W \cup RE_1 \cup RE_2$, where $W$ = {abededbc, bedad, cedabc} and $RE_1$ = ab*edab, $RE_2$ = abe{3, 5}d*ca{2,6}bd.

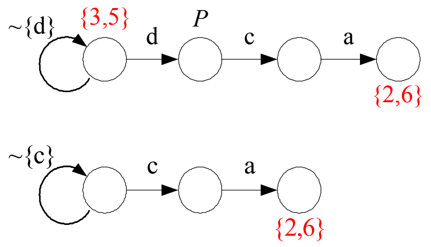
Figure 2 shows the resulting goto graphs. If $x$ = cabebdabedaacafabde, then "abebdabedaacababd" is identified when the last "d" is processed. Note that a forked traversal is created when the first "e" is processed and another forked traversal is created when the second "e" is processed. The first forked traversal finds the match and the second forked traversal ends when "f" is processed.



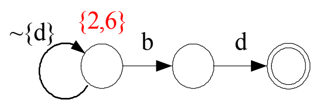
(a) The goto graph G'.



(b) The goto graph D'.



(c) The goto graph for the second fragment of $RE_2$.



(d) The goto graph for the third fragment of $RE_2$.
Figure 2. The goto graphs for Example 2.

## V. Conclusion

We have presented in this paper a systematic approach to construct the finite state pattern matching machine for a set of strings together with some simple regular expressions. Like the Aho-Corasick algorithm, our proposed construction procedure yields a pattern matching machine dictated by three functions, namely, the goto, failure, and output functions. The difference is that our constructed pattern matching machine may consist of multiple separated goto graphs connected by failure functions. Theoretically, there could be a large number of forked traversals for a regular expression which consists of multiple {*min*, *max*} operators. In practice, there should not be a larger number of forked traversals for clean traffic, as long as the first fragment is sufficiently long. Therefore, in real applications, one can set a limit, say 8, on the number of concurrent forked traversals for each traffic flow. Comparison of the performance of our proposed pattern matching machine with that of the ClamAV implementation will be reported in a future paper.

## References

[1] Clam anti virus signature database, www.clamav.net.
[2] D. E. Knuth, J. H. Morris, and V. R. Pratt, "Fast pattern matching in strings," TR CS-74-440, Stanford University, Stanford, California, 1974.
[3] R. S. Boyer and J. S. Moore, "A fast string searching algorithm," Communications of the ACM, Vol. 20, October 1977, pp. 762-772.
[4] A. V. Aho and M. J. Corasick, "Efficient string matching: an aid to bibliographic search," Communications of the ACM, Vol. 18, June 1975, pp. 333-340.
[5] S. E. Schechter, J. Jung, and A. W. Berger, "Fast detection of scanning worm infections." 7th International Symposium on Recent Advances in Intrusion Detection, French Riviera, September 2004.