Intelligent Grouping Algorithms for Regular Expressions in Deep Inspection

Zhe Fu[*†], Kai Wang[*†], Liangwei Cai[§], and Jun Li[†‡]
[*]Department of Automation, Tsinghua University, Beijing, China
[†]Research Institute of Information Technology, Tsinghua University, Beijing, China
[§]College of Information Engineering, Shenzhen University, Shenzhen, Guangdong, China
[‡]Tsinghua National Lab for Information Science and Technology, Beijing, China
{fu-z13, wang-kai09}@mails.tsinghua.edu.cn, cailw@szu.edu.cn, junl@tsinghua.edu.cn

Abstract—**Deep inspection is widely used to identify network traffic. Due to the complexity of payload in traffic, regular expressions are becoming the preferred choice to specify the rules of deep inspection. Compiling a set of regular expressions into one Deterministic Finite Automata (DFA) often leads to state explosion, which means huge or even impractical memory cost. Distributing a set of regular expressions into multiple groups and building DFAs independently for each group mitigates the problem, but the previous grouping algorithms are either brute-force or locally optimal and thus not efficient in practice. Inspired by the Intelligent Optimization Algorithms, we propose new grouping algorithms based on Genetic Algorithm and Ant Colony Optimization algorithm, to effectively solve the problem of state explosion by acquiring the global optimum tradeoff between memory consumption and the number of groups. Besides, to accelerate the execution speed of the intelligent grouping algorithms, we employ and improve an approximation algorithm that estimates the DFA states according to the conflicting relationship between each pair of regular expressions. Experimental results verify that our solutions save around 25% memory consumption or reduce around 20% of group number compared with existing popular grouping algorithms.**

Keywords—Deep Inspection; Regular Expression Grouping; DFA; Intelligent Optimization

I. INTRODUCTION

Deep Inspection (also known as content scanning or payload scanning of network traffic) is now playing a key part in network security. The payload of packet streams in network traffic is scanned against a set of rules, which describes the signatures that characterize the threats (e.g. spam, virus or anonymous intrusion), to identify the network flows with specific data in the payload of their packets.

Due to its expressiveness and flexibility in describing payload signatures, regular expression is widely used in deep inspection tools, including popular open source software Snort [1], L7-filter [2] and Bro [3]. There are two kinds of finite automata (FA) used to implement the regular expression matching: Nondeterministic Finite Automata (NFA) and Deterministic Finite Automata (DFA), and each of them has its strengths and weaknesses. NFA has less states and transitions, and its space cost linearly depends only on the size of regular expression rule set, thus it is space-efficient. A theoretical study [4] shows that it could generate an NFA with $O(m)$ states for a single regular expression whose length is m. However, in the worst case, there can be $O(n)$ state traversals per input character for an NFA with n states, which may require tremendous memory bandwidth, so that NFA is very slow. On the contrary, DFA only activates one state and requires exactly one state traversal per input character. As a result, DFA is fast, and thus it is the preferred choice to implement deep inspection.

However, the time-efficiency of DFA is at the cost of huge amount of memory usage. Compared to NFA, DFA needs more states and transitions, and in many cases suffers from the state explosion. Yu et al. [5] discussed five categories of regular expressions used in deep inspection application that can cause blowing up of corresponding DFA. One reason is concluded as the interactions among regular expressions. In general, for the regular expressions with interactions, the DFA constructed from them together has a far larger size than the sum of the sizes corresponding to the DFAs built from each regular expression individually. For example, when compiling regular expressions `<.*ab.*cd>` and `<.*ef.*gh>` into a composite DFA, the number of states can be doubled as shown in Fig. 1.

To address the problem of state explosion, we propose two intelligent grouping algorithms. In detail, our contributions can be concluded as follows.

- The grouping problem is formulated into two separate cases according to practical demands: (1) given the number of groups, minimize the number of DFA states and (2) given upper limit of DFA state number, minimize the number of groups. Two intelligent grouping algorithms are proposed inspired by Genetic Algorithm and Ant Colony Optimization algorithm, and effectively solve the problem of state explosion by acquiring the global optimum distribution in each situation.
- The method to estimate the number of DFA states are improved so as to accelerate the executing speed of the intelligent grouping algorithms, basing on previous work that use the interaction of regular expressions pairs.
- The performance of the proposed algorithms is compared to the previous grouping algorithms for

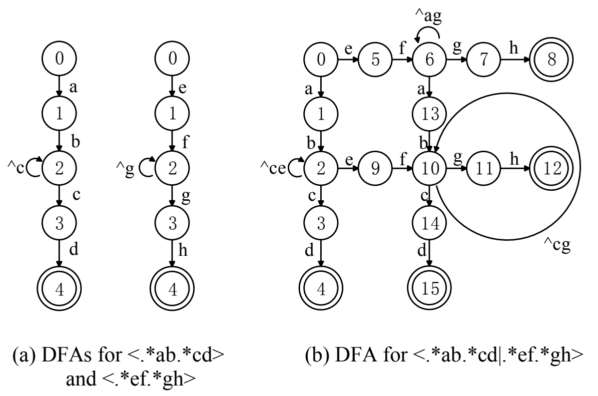

(a) DFAs for <.*ab.*cd> and <.*ef.*gh>
(b) DFA for <.*ab.*cd|.*ef.*gh>

Fig. 1. Example of state explosion

evaluation. Experimental results based on practical rule sets show that intelligent grouping algorithms save around 25% memory overhead or reduce around 20% amount of groups.

This paper is organized as follows. In Section II, we present the related work. Section III categorizes the grouping problem into two kinds of situations on demand, and Section IV puts forward two intelligent grouping algorithms. In Section V, further optimization, including an improved method to estimate the number of DFA states, are proposed. Experimental results are shown in Section VI. In the last section, we give our conclusion.

II. Related Work

Currently, several techniques for reducing the memory consumption of DFAs constructed from regular expressions have been proposed.

DFAs Compressions

Kumar et al. invented D^{2}FA (Delayed input DFA) [7] to reduce the amount of transitions by adding a default transition, and it diminishes the memory occupied by DFAs to represent the regular expressions. However, it is at the expense of more memory accesses overhead, because multiple default transitions will be executed when no corresponding transitions are found in the compressed states for certain input character. The D^{2}FA-derived algorithms (like δFA [8] etc.) are all designed for further improving the matching speed or compression ratio. Smith et al. [9] proposed XFA (eXtended Finite Automata), which reduces the number of states by leveraging flag/counter variables to replace the redundant states. However, the introduced variable fetching and calculation will cause memory bandwidth and I/O bottleneck. Other techniques of compressing DFAs' space consumption such as state merging [10], H-FA [11], hybrid FA [12] similarly reduce memory at the cost of more matching time and worse temporal efficiency.

Regular Expressions Rewrites

Yu et al. [4] analyzed the features of regular expressions that cause the explosions of states, and proposed two rewriting rules, one for patterns that generate quadratic-sized DFAs, and the other for those generating exponential-sized DFAs. Although the sizes of DFAs constructed from rewritten regular expressions can be reduced from quadratic or exponential to only linear [5], the technique applies only to regular expressions

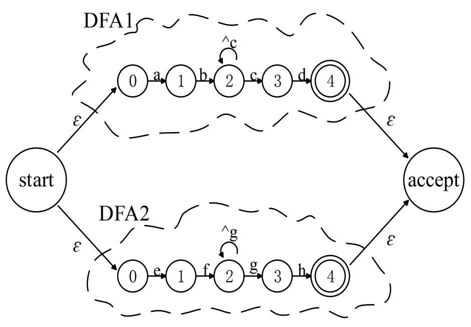


Fig. 2. Composing two DFAs

with specific patterns, thus it cannot handle all state explosion problems of various regular expressions.

Hardware-based Techniques

FPGA (Field-Programmable Gate Array) and TCAM (Ternary Content Addressable Memory) devices have recently gained much attention to improve the executing efficiency of DFAs [13-15]. FPGA devices have advantages in the implementation with pipeline and parallelism, however, the small size of on-chip memories of FPGA limits the practical deployment of large-scale rule sets. TCAM devices are fast, but the updating algorithm may cost large amount of storage space. Worse more, the high cost and big power consumption make FPGA and TCAM devices not cost-effective for regular expressions matching in deep inspection.

Regular Expression Grouping

In [5], Yu et al. put forward the ideas of greedily grouping regular expressions to construct multiple automata, and conceived and implemented a grouping algorithm only using the information whether the regular expressions have interactions (if the composite DFA of two regular expressions generates more states than the sum of the number of states in the DFAs individually constructed from each of them, they may be called "interactive"), without considering how much they interact with each other. Moreover, the grouping algorithm simply picks one ungrouped regular expression and places it in the group where it has the least interactions with others. Conclusively, this algorithm may trap into local optimum easily.

Rohrer et al. [6] evaluated the "distance" between each pair of regular expressions, and then converted the distribution problem to the Maximum Cut problem. Several methods including Poles Heuristic and Simulated Annealing are leveraged as the solutions that make full use of the relationships among the regular expressions, but generally only finding sub-optimal solutions within a reasonable run time. Besides, the mathematical model used to convert the grouping algorithm to Maximum Cut problem still has to be improved, otherwise it will affect the final result as described in detail in Section V.

III. Problem Formulation

As a simple example for grouping regular expressions shown in Fig. 2, if the rule set, which has two regular expressions `<.*ab.*cd>` and `<.*ef.*gh>`, is divided into two individual groups (each group has one regular expression), the total number of states will be reduced from 16 to 10. Although the state explosion problem can be efficiently mitigated by regular expression grouping, it does not make much sense to focus only on the total number of DFA states or the number of groups derived from the grouping algorithm. Minimizing only the number of states will end up with large subdivided group count, and minimizing only the number of groups will obviously get back to state explosion. In fact, the two metrics used to evaluate the performance of regular expression grouping method should be considered according to different practical demands.

A. Given the number of groups, minimize the number of DFA states

In practice, the group number is limited by the maximum number of DFAs a particular hardware platform supports to execute in parallel. If the number of groups is greater than that of the calculation cores/threads, i.e., generating more DFAs than the calculation cores/threads, the latency of parallel processing may make this situation time-inefficient. On the other hand, if the DFAs generated are fewer than the calculation cores/threads, several of the cores/threads are unoccupied, so that it fails to make full use of computational resource. Therefore, in the case of parallel devices with k cores/threads available or allocated for regular expression matching, the optimal number of groups (DFAs) is k.

In this case, the target problem is to distribute a set of regular expressions into k groups, where each subset constructs a DFA, and the total state number of all DFAs is minimal. In mathematics, the problem can be formally defined as follows:

For a regular expression set $S = \{r_1, r_2, \dots, r_i, \dots\}$, $i \in [1, N]$ and a given k, the aim is to find k disjoint subsets $R_1, R_2, \dots, R_k$, $\bigcup_{m=1}^{k} R_m = S$, to minimize the overall memory consumption $\sum_{m=1}^{k} T(R_m)$, i.e., minimize $\sum_{m=1}^{k}(T(\sum_{r_i \in R_m} r_i))$, where $T(R_m) = (T(\sum_{r_i \in R_m} r_i))$ represents the number of states of the DFA constructed by subset R_j.

In case of $k = 2$, there exists $\prod_N 2 \,/\, 2! = 2^{N-1}$ distinct possible distributions. By analogy, for a set of N regular expressions, which is to be divided into k groups, there exists $k^N/k!$ distinct solutions [6]. The exponential growth makes the number of possible distributions too enormous to find the optimal distribution. Take a practical rule set used in deep inspection for example. A rule set from L7-filter is composed of 111 regular expressions, if distributed into two groups, the number of distinct possible distributions is $2^{111}/2! = 2^{110} \approx 10^{33}$. Thus, it is obviously infeasible to use brute-force calculation to find the optimal or close-to-optimal distribution from all possible solutions.

B. Given upper limit of DFA state number, minimize the number of groups

In some cases, the memory space taken by the states is limited by physically available memory size of a particular hardware platform (such as certain embedded devices), which results in the fact that the memory consumption, rather than the number of groups, is the key point for deep inspection implementation. Under given memory limit, the aim is to acquire as few groups as possible, to reduce the latency of throughput and accelerate the matching speed.

In this paper, the State Expansion Ratio (SER) is defined as the metric to quantify the memory cost. For a regular expression set $S = \{r_1, r_2, \dots, r_i, \dots\}$, $i \in [1, N]$, the SER is defined as bellows:

$$SER = \frac{T(\sum_{r_i \in S} r_i)}{\sum_{r_i \in S} T(r_i)} \tag{1}$$

where $T(r_i)$ represents the number of states of the DFA constructed from regular expression r_i. Obviously, large SER indicates serious state explosion in set S. Then the problem to optimize grouping based on limited number of states can be formally defined as follows:

For a regular expression set $S = \{r_1, r_2, \dots, r_i, \dots\}$, $i \in [1, N]$ and a given SER limit THR, the aim is to find k disjoint subsets $R_1, R_2, \dots, R_k$, $\bigcup_{m=1}^{k} R_m = S$, where k is minimal. To minimize k, and $\forall \mathrm{m} \in [1, k]$, there exists:

$$\frac{T\left(\sum_{r_i \in R_m} r_i\right)}{\sum_{r_i \in R_m} T(r_i)} \le THR \tag{2}$$

In order to reduce memory consumption, the secondary goal under this situation is to reduce total number of DFA states corresponding to the set S as much as possible on the premise that k is minimal.

IV. Algorithm

Intelligent Optimization Algorithms (IOAs) are popular in solving many optimization problems, such as Traveling Salesman Problem (TSP), (Bounded) Knapsack Problem (BKP), and so forth. Typical IOAs include Simulated Annealing (SA), Genetic Algorithm (GA), Tabu Search (TS), Ant Colony Optimization (ACO), etc.

As a matter of fact, regular expression grouping problem can be transformed into an optimization problem. An accurate optimum solution can be obtained easily by iteratively checking every possible optimization result for a small-scale optimization problem. However, as mentioned in Section III, for a set of N regular expressions which is to be divided into k groups, brute-force searching for optimum is impractical due to the possible $k^N/k!$ distinct distributions.

Section III categorizes the grouping problem into two separate cases according to practical needs. Correspondingly, in this Section we propose two intelligent grouping algorithms GABG (Grouping Algorithm Based on Gene) and GABAC (Grouping Algorithm Based on Ant Colony), to treat each case.

A. GABG -- Given the number of groups, minimize the number of DFA states

1) Introduction to GA

Genetic Algorithm (GA), a search heuristic that mimics the "survival of the fittest" process of natural evolution, is first proposed by John Henry Holland in 1975 [16], and rapidly applied to the fields of combinatorial optimization, machine learning, etc. Darwin's theory of natural selection explains that biological evolution is the result of heritable variation and survival struggle. As long as there is a slight variation among the population, the environment will select the most advantageous individuals inevitably. The process of variation, selection and

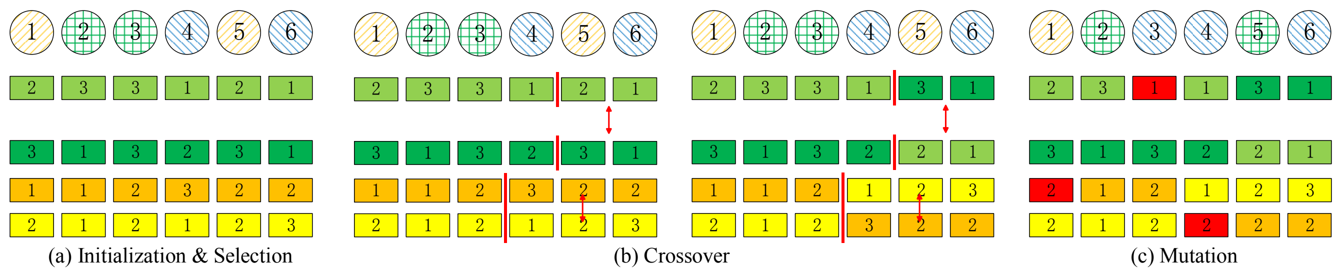


Fig. 3. A simple example of using GABG to solve the grouping problem of 6 regular expressions to be distributed into 3 groups, with only 4 individuals

inheriting will lead to a progressive evolution, and only the fittest individuals will survive.

In general, a Genetic Algorithm has five steps.

Step1: Initialization. Certain individuals are usually randomly generated as an initial population, allowing the entire range of the search space.

Step2: Evaluation. Calculate the fitness of all the individuals according to the objective function.

Step3: Selection. Individuals are selected by a probabilistic method, where fitter individuals are more likely to be selected.

Step4: Crossover and Mutation. Cross the chromosomes of distinct individuals selected in Step3 and then mutate.

Step5: Termination. If a termination condition is reached, the generational process terminates; otherwise, go to Step2.

It is nontrivial to note that, we will follow these five steps above to design the algorithm framework.

2) Design and implementation of GABG

In this case, the number of groups is given, which is similar to the genes on chromosomes. In classical GA, the chromosomes of individuals are represented in binary as strings of 0s and 1s, yet this situation is not suitable for regular expression grouping problem. Thus, we propose an index structure instead of the common binary chromosome. Corresponding to biological concepts, for a problem of distributing a set of N regular expressions into k groups, we define as follows:

- **Chromosome:** representing the distribution of current individual. The length of all chromosomes is N.
- **Gene:** the bit in a chromosome, ranging from 1 to k. The genes on the chromosome are in orderly rows and the value of each gene represents the serial number of which the corresponding regular expression is distributed into.
- **Individual:** in addition to one chromosome, every individual contains the fitness that is calculated according to the number of total states (negative correlation) and the probability this individual will be selected.
- **Population:** collection of a number of individuals.

Figure 3 depicts a simple example of using GABG to distribute 6 regular expressions into 3 groups, on the condition of only 4 individuals. After initialization, each regular expression is distributed randomly. Each color of the circles represents the serial number that the corresponding regular expression distributed into. As shown in Fig. 3a, No.4 and No.6 of the regular expressions are in the same color, which indicates that No.4 and No.6 are to be distributed into the same group (named Group 1), so the values of genes under the circles are 1. By such analogy, No.1 and No. 5 are initially distributed into Group 2, while No.2 and No.3 are distributed into Group 3. That is the initiation of the first individual.

The other 3 individuals are initiated in the same way. Then calculate the fitness of this population, and select individuals according to Roulette Wheel Selection [17]. As in Fig. 3b, we select the first and second individual as an example, then choose a position randomly and crossover the two chromosomes at p_c (probability of performing crossover). This operation is also performed for the remaining two individuals. After that, pick a bit on the chromosome randomly and mutate it into arbitrary value between 1 and k at p_m (probability of mutation). Based on the above steps, the algorithm generates a new population with four different individuals, as shown in Fig. 3c.

Using Finite Markov Chains, Rudolph et al. [18] proved that the Canonical Genetic Algorithm with the process of crossover, mutation, and selection operations cannot guarantee the convergence to global optimum. For this reason, the GABG adopts an elitist strategy which maintains the best solution found over time before next-round selection, i.e., find the worst individual in current generation and replace it with the best individual from the last generation. Rudolph et al. [18] proved its global convergence in theory when this strategy is taken.

The pseudo-code of GABG is as below. The function `rand()` generates a random number between 0 and 1.

Algorithm 1 – GABG

```
function GABG(RE set)
  Initialize();
  while(cnt++ < MAXGEN)  do
    Cal_fitness();
    Best_individual ← find_best();
    while( k++ < SIZE/2 )
      individual_i ← Select();
      individual_j ← Select();
      if(rand()<p_cross) then
        cross(individual_i, individual_j);
      end if
      if(rand()<p_mutate) then
        mutate(individual_i);
      end if
      if(rand()<p_mutate) then
        mutate(individual_j);
      end if
    end while
    Worst_individual ← find_worst();
    Worst_individual ← Best_individual
    cal_fitness();
  end while
  return find_best();
end function
```

The operations of crossover and mutation (step 9 – step 17) guarantee that the algorithm will not fall into local optimum, meanwhile the elitist strategy (step 5, step 19 and step 20) accelerates the convergence speed.

B. GABAC -- Given upper limit of DFA state number, minimize the number of groups.

1) Introduction to ACO

Ant Colony Optimization algorithm (ACO) is a probabilistic method to solve the problem of finding good paths in a graph, inspired by the behavior of ants seeking food in the natural world. In 1992, Marco Dorigoet al. [19] initially proposed the first ACO algorithm. Since then it has been applied to many combinatorial optimization problems, such as Job-shop Scheduling Problem (JSP), TSP, and so on.

It is discovered by biologists that the low-intelligent ant individuals show high intelligence, i.e., self-organization, in a colony without centralized command. When seeking food, ants wander randomly at first, laying down several pheromones, which will influence the choices of other ants. Meanwhile, the pheromones evaporate at a certain rate, thus the shorter the path is, the less time it takes for an ant to travel from its colony to the source of food and return back, the more pheromones remain and the more likely for other ants to follow this path. The positive feedback by the pheromones eventually leads to a single path which all the ants follow, and the probabilistic wandering and evaporation of pheromones avoid the convergence to a locally optimal solution.

2) Design and implementation of GABAC

The general ACO algorithm aims at finding the best path to link a series of cities, such as TSP. The aim of this case is to find minimum number of groups, which is much like a TSP problem. However, ACO algorithm cannot be applied to regular expression grouping problem directly. The proposed GABAC algorithm takes advantage of the two core ideas in ACO, namely the probabilistic wandering and positive feedback.

Probabilistic Wandering

Whether two or more regular expressions are grouped together will be determined according to the estimated probability. Compared to the upper limit THR, the smaller the SER is, the greater the probability being in the same group should be. Further, the function of probability is conceived non-linear so that the regular expressions causing less state explosion will have much greater chances to be put together in one group.

For a regular expression set $S = \{r_1, r_2, \dots, r_i, \dots\}$, $i \in [1, N]$, meanwhile the upper limit of SER is THR, the probability of adding regular expression r_i into the subset R_m is: (pheromones omitted)

$$P_{add}(R_m, i) = \frac{\log((\sum_{r_k \in R_m} T(r_k) + T(r_i)) \times THR) - \log(T(\sum_{r_k \in R_m} r_k + r_i))}{\log(THR)} = 1 - \frac{\log(SER)}{\log(THR)} \quad (3)$$

where $T(r_i)$ represents the number of states of the DFA constructed by regular expression r_i. Under normal circumstances, $P_{add}(R_m, i) \in (0,1)$.

Positive Feedback

$pherpmone[i][j]$, ranging from 0 to 1, stands for the interrelationship between two regular expressions r_i and r_j. Large pheromones mean that r_i and r_j have been distributed into the same group repeatedly, which indicates that r_i and r_j have a high tendency to be grouped together. The improved formula of calculating the probability of adding regular expression r_i into the subset R_m is:

$$P'_{add}(R_m, i) = (1 - \alpha) P_{add}(R_m, i) + \alpha \frac{\sum_{r_k \in R_m} pherpmone[i][k]}{num\_of(R_m)} \quad (4)$$

where α is the weight of pheromones, and $\frac{\sum_{r_k \in R_m} pherpmone[i][k]}{num\_of(R_m)}$ means the average amount of pheromones between regular expression r_i and the group R_m. It is clear that more pheromones lead to a higher probability of adding r_i into R_m. If r_i is added into the group R_m, the pheromones between r_i and each regular expression of R_m increase as follows:

$$\Delta pherpmone[k][i] = P_{add}(R_m, i) \times \beta, \forall r_k \in R_m \quad (5)$$

where β is the weight of pheromones an ant lays down. Only thing to note here is that the maximum value of pheromones is 1, otherwise the pheromones will increase unquenchably even if there exists a process of evaporation.

After all ants reach the destination, the pheromones evaporate as below:

$$pherpmone[i][j] = pherpmone[i][j] \times (1 - \gamma), \forall r_i, r_j \in S \quad (6)$$

where γ is the weight of evaporation.

Based on the formulas above, for a regular expression set $S = \{r_1, r_2, \dots, r_i, \dots\}$, $i \in [1, N]$ and the upper limit of SER is THR, assuming that there exists m ants, the process of GABAC is:

Initially, each of the m ants selects one regular expression randomly as the start of grouping. Then each ant randomly selects one of the ungrouped regular expressions, and calculates the probability of adding it to the current group. If the condition of grouping is met, add this regular expression into the current group and increase the pheromones between this regular expression and each of the current group. Otherwise, repeat the previous steps until one regular expression is added. If no regular expression meets the condition of adding into the current group, select one randomly from the ungrouped regular expressions as the start of next-round grouping. When all m ants achieve the destination (i.e., the GABAC algorithm obtains m solutions), record the best solution, evaporate the pheromones and place all m ants at the starting point (i.e., each ant selects one regular expression randomly as the start of grouping again). After deterministic iterations, the GABAC algorithm terminates. One can find that the final solution is just the optimum of the superior solutions recorded in each round iteration.

The pseudo-code of GABAC is as below:

Algorithm 2 – GABAC

```
1  function GABAC(RE set)
2    while(cnt++ < MAXGEN)  do
3      put all ants at the start;
4      for each ant[i] in the colony
5        add a random RE to ant[i] as 1st group;
6      end for
```

```
    while(!all_ants_have_no_space_to_go) do
      for each ant[i] in the colony
        for each RE[j] remained for ant[i]
          if P'add(ant[i],RE[j])>rand())then
            add RE[j] to ant[i];
            increase pheromones;
            break;
          end if
        end for
        if every remaining RE for ant[i] is examined
          add a random RE to ant[i] as another group;
        end if
      end for
    end while
    result ← best ant;
    evaporate();
  end while
  return best of results;
function end
```

The merit of grouping regular expressions with probabilities lies in the avoidance of the convergence to a locally optimal solution, and the introduction of pheromones facilitates the positive feedback and accelerates the convergence speed.

V. Optimization

Because intelligent grouping algorithms (both GABG and GABAC) need a lot of iterative computations, calculating the accurate number of total states by building DFA in each-round iteration is impractical. The shortcomings are twofold:

Time-inefficient. Through testing, for the problem of distributing 111 regular expressions into 4 groups with given 50 individuals and 5000 generations in GABG, it takes nearly 20 hours to obtain a final solution using the Regular Expression Processor [20], and the time cost may increase exponentially with the number of regular expressions and groups that regular expressions are distributed into.

Unreachable. Due to the randomness, both GABG and GABAC may encounter the situation that the construction of a temporal DFA which is explosive will be infeasible in the actual platform, and the number of states in such a case cannot be calculated.

Rohrer et al. [6] formulated the distribution problem as an energy minimization problem. His work first defined the coefficient a_i as the number of states of the DFA constructed by regular expression r_i itself, then defined the coefficient $b_{i,j}$ as the number of states of the DFA constructed by regular expression pair r_i and r_j. On this basis, his work defined the mutual distance between regular expression r_i and r_j as $m_{i,j} = b_{i,j} - a_i - a_j$, which means the increase in the number of states when compiling r_i and r_j into a composite DFA compared to the sum of the individual DFA of r_i and r_j. Rohrer et al. made a assumption that when adding a third regular expression r_l to the pair of r_i and r_j, the increase in number of states of the composite DFA is $m_{il} + m_{jl}$. On the basis of above-mentioned assumption, for a regular expression set $S = \{r_1, r_2, \dots, r_i, \dots\}$, $i \in [1, N]$ which is to be distributed into k groups, the number of states can be evaluated by the following formula:

$$E(S) = \sum_{m=1}^{k} E(R_m) = \sum_{m=1}^{k} \sum_{r_i \in R_m} a_i + \sum_{m=1}^{k} \sum_{r_{i,j} \in R_m, i<j} m_{i,j}$$

$$= \sum_{i=1}^{N} a_i + \sum_{m=1}^{k} \sum_{r_{i,j} \in R_m, i<j} m_{i,j} \qquad (7)$$

where R_m represents each subset according to the distribution and $E(R_m)$ represents the number of states of the DFA constructed by regular expressions subset R_m. Rohrer et al. [6] simply used the formula (7) to estimate the sum of states of all the DFAs, or the memory consumption in other words, when compiling the set of regular expressions into the same DFA.

The experimental results (shown in Section VI) point out that although the formula (7) can be an applicable approximation on the condition of a small quantity of regular expressions or slight state explosion in the regular expression set, the increase of accurate number of states is far greater than the approximated number of states in other cases which will cause remarkable error for the optimization goals.

On the basis of Rohrer's work, we define another coefficient $\rho_{i,j}$:

$$\rho_{i,j} = \frac{b_{i,j} - a_i - a_j}{a_i + a_j} = \frac{m_{i,j}}{a_i + a_j} \qquad (8)$$

as the expansion rate of regular r_i and r_j, $\rho_{i,j} \geq 0$, $\forall i,j \in \{1, N\}$.

The increase of approximate number of states is not only relevant to the mutual distance $m_{i,j}$, but also the expansion rate $\rho_{i,j}$, especially in the situation that there exists relatively huge number of regular expressions or serious state explosion. When adding a third regular expression r_a into the group R_m, according to the improved approximation, the increase of the number of states is:

$$\Delta E = \sum_{r_i \in R_m} \left(\left(\sum_{r_j \in R_m, j \neq i} (\rho_{a,j} + \rho_{i,j}) \right) \times m_{a,i} \right) \qquad (9)$$

Then the sum of states of group R_m when compiled into the same DFA is:

$$E(R_m) = \sum_{r_i \in R_m} a_i + \sum_{r_{i,j} \in R_m, i<j} \sum_{r_l \in R_m, l \neq i,j} (\rho_{i,l} + \rho_{j,l}) \times m_{i,j} \qquad (10)$$

For a regular expression set $S = \{r_1, r_2, \dots, r_i, \dots\}$, $i \in [1, N]$ which is to be distributed into k groups, the improved estimation is:

$$E(S) = \sum_{m=1}^{k} E(R_m)$$

$$= \sum_{m=1}^{k} \sum_{r_i \in R_m} a_i + \sum_{m=1}^{k} \sum_{r_{i,j} \in R_m, i<j} \sum_{r_l \in R_m, l \neq i,j} (\rho_{i,l} + \rho_{j,l}) \times m_{i,j}$$

$$= \sum_{i=1}^{N} a_i + \sum_{m=1}^{k} \sum_{r_{i,j} \in R_m, i<j} \sum_{r_l \in R_m, l \neq i,j} (\rho_{i,l} + \rho_{j,l}) \times m_{i,j} \qquad (11)$$

In Section VI a comparison experiment between the energy function and the improved estimation is made. The result shows that the improved method using the additional coefficient $\rho_{i,j}$ results in far better approximation in both numeric and variation tendency.

VI. Evaluation

In this section, we conduct a series of experiments to evaluate the effectiveness of our intelligent grouping algorithms and the optimization. Experiments are conducted on an Intel(R) Xeon(R) E5335 platform (CPU: 2.0GHz, Memory: 4GB, OS: Ubuntu 12.04 64bit). The Regular Expression Processor [20] is used to calculate the accurate number of states of a regular expressions distribution, as the metric of memory consumption.

Five rule sets picked from open source software and two from commercial corporations are used, as shown in Table I.

TABLE I. RULE SETS INFORMATION

name	*number of REs*	*source*	*interaction*
linux-111	111	L7-filter	moderate
snort-120	120	Snort	serious
l7filter-111	111	L7-filter	moderate
bro-1196	1196	Bro	serious
snort-1190	1190	Snort	serious
app1-1386	1386	commercial use	serious
app2-5886	5886	commercial use	moderate

A. Improved approximation evaluation

In section V an improved estimation of state numbers based on work [6] is proposed. It is impossible to compare every number of states of all possible distributions between exact value and approximation, therefore we continually add one regular expression from a rule set to a test group randomly, and calculate the number of states of the DFA constructed by test group.

Figure 4 depicts the total number of states when the number of regular expressions in test group is increasing. It is shown that if the test group contains fewer regular expressions (less than 15), both of the original approximation and the improved approximation are not far from the exact valve. However, if more regular expressions are added to the test group, the improved approximation and the exact value increase followed the same trend, while the original approximation increases too slowly. When adding 22 regular expressions to the test group, the state number of the improved approximation as well as that of exact value is around 5 times as huge as that of the original approximation. As one can see, in both numeric and tendency, the improved method which adds the coefficient $\rho_{i,j}$ leads to far better approximation than the original approximation in Rohrer's work [5].

B. Evaluation of GABG

This subsection shows the experimental results under the circumstance of a given number of groups. We implement Simulated Annealing (SA) and Poles heuristic (poles) algorithms [6] as comparisons. In Fig. 5, the histograms show the absolute figure of state numbers achieved by GABG, SA and Poles heuristic corresponding to the given number of groups (from 2 to 6) for rule sets snort-120 and linux-111, and the

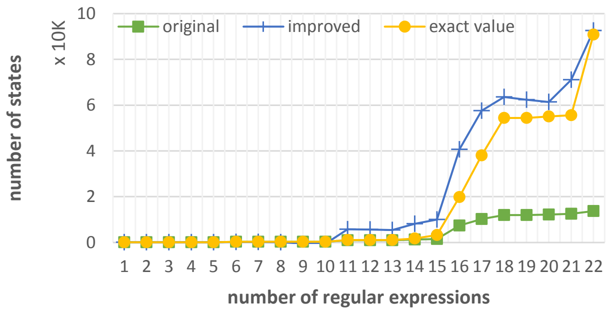


Fig. 4. States number comparison on linux-111

broken line graph presents the relative proportions of results achieved by GABG and SA compared to Poles heuristic. As one can see, GABG achieves better distribution results than SA and poles.

After making statistics and analysis of multiple grouping results for different rule sets, it is concluded that GABG saves 12% at least and 25% on average memory consumption compared to Simulated Annealing algorithm, or around 60% memory consumption compared to Poles heuristic, since GABG can find optimal solution in each situation, while SA and poles can only achieve sub-optimal results.

C. Verifying the effects of pheromones

GABAC has two specialties: probability selection according to SER and positive feedback by pheromones. It is not clear that whether the introduction of pheromones takes effect in accelerating the convergence speed, thus we make an experiment to verify it. The population size of ants is set as 100 and the iterations are limited to 50. Depending on different THR(from 3 to 8), the numbers of groups of GABAC with and without pheromones are depicted in Fig. 6.

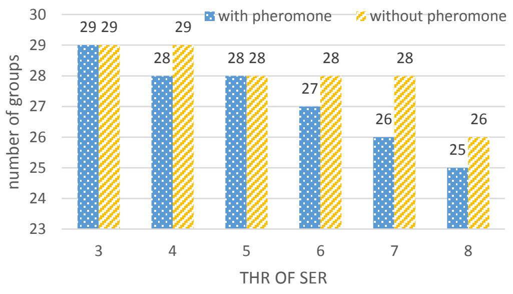
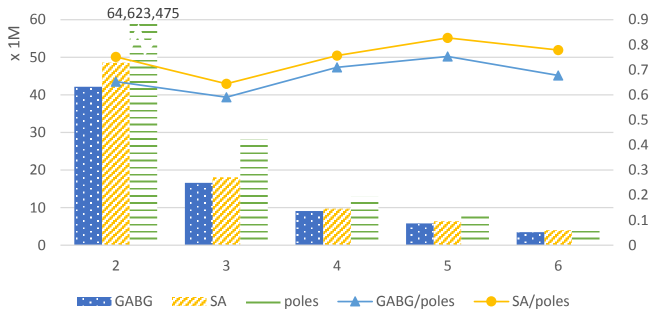
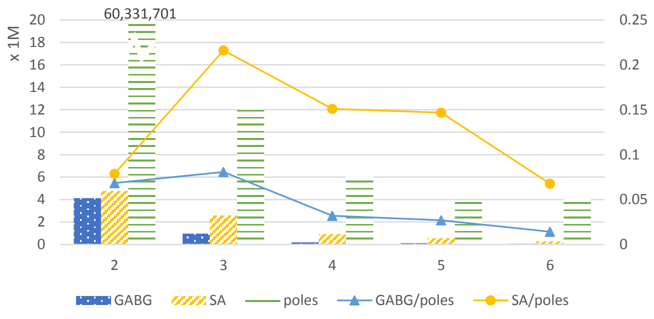


Fig. 6. Verifying the effects of pheromones (snort-120)

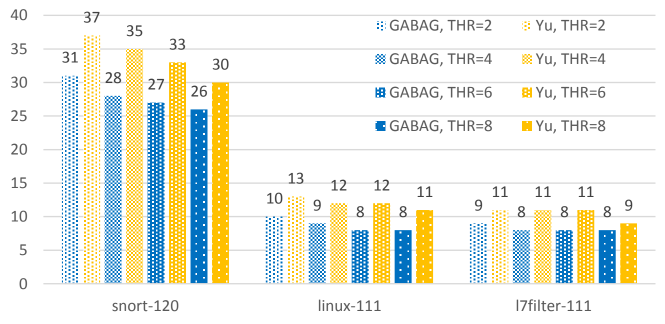


(a) numbers and proportions of states (snort-120)

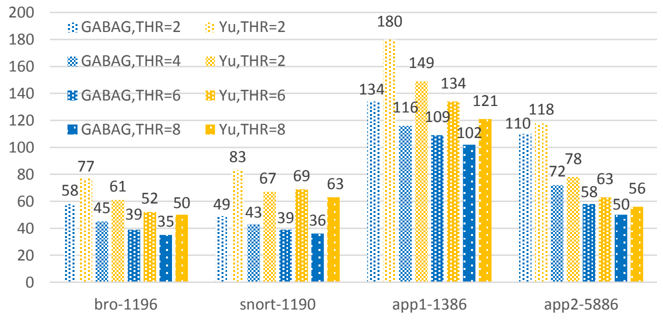


(b) numbers and proportions of states (linux-111)

Fig. 5. Experimental results of GABG



(a) numbers and proportions of states (small-scale rule sets)



(b) numbers and proportions of states (big-scale rule sets)

Fig. 7. Experimental results of GABAC

With the same parameters, the results of GABAC with pheromones are obviously better than that without pheromones. Therefore, we may draw a conclusion that the pheromones positively work in GABAC.

D. Evaluation of GABAG

When the upper limit of SER is given, the aim is to find a distribution that generates as least groups (DFAs) as possible, and reduce the number of total number of DFA states if the group number reaches a minimum. We experiment on small-scale rule sets (snort-120, linux-111 and l7filter-111) and large-scale rule sets (bro-1196, snort-1190, app1-1386 and app2-5886). Grouping algorithm proposed by Yu et al. [5] is implemented as comparison.

As shown in Fig. 7, we depict the number of groups depending on different THR (2, 4, 6 and 8). For all rule sets, GABAC achieves fewer groups in each situation, reducing around 20% of group number by acquiring the global optimum solution of distribution compared to Yu's algorithm.

It must be noted that both GABG and GABAC spend more time compared with existing grouping algorithms. However, the grouping algorithm runs offline so its complexity is not a major concern.

VII. Conclusion

In this work, we propose and evaluate two intelligent grouping algorithms based on Genetic Algorithm (GABG) and Ant Colony Algorithm (GABAC), aiming at solving the problem of DFA state explosion. We also make an optimization on the object function of proposed algorithms to accelerate the execution speed. The experimental results on practical rule set of regular expressions draw the preliminary conclusion that the proposed algorithms save around 25% memory consumption compared to Simulated Annealing algorithm and around 60% memory consumption compared to Poles heuristic, or reduce 20% of group number compared to Yu's algorithms, by acquiring the global optimum solution of distribution. Our future work will focus on the preliminary distribution of regular expressions according to their syntaxes, which can further speedup the convergence of the grouping procedure.

References

[1] Snort, http://www.snort.org/.

[2] L7-filter, http://l7-filter.sourceforge.net/.

[3] Bro, http://www.bro.org/.

[4] J. E. Hopcroft and J. D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison Wesley, 1979.

[5] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman and R. H. Katz, Fast and Memory-Efficient Regular Expression Matching for Deep Packet Inspection, in Proc. of ACM/IEEE ANCS, 2006.

[6] J. Rohrer, K. Atasu, J. V. Lunteren and C. Hagleitne, Memory-Efficient Distribution of Regular Expressions for Fast Deep Packet Inspection, in Proc. of the 7th IEEE/ACM international conference on hardware/software codesign and system synthesis, 2009.

[7] S. Kumar, S. Dharmapurikar, F. Yu, P. Crowley and J. Turner, Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection, in Proc. of ACM SIGCOMM, 2006.

[8] D. Ficara , S. Giordano, G. Procissi, F. Vitucci, G. Antichi and A. Di Pietro, An Improved DFA for Fast Regular Expression Matching. ACM SIGCOMM Computer Communication Review, 38(5), 29-40, 2008

[9] R. Smith, C. Estan and S. Jha and S. Kong, Deflating the Big Bang: Fast and Scalable Deep Packet Inspection with Extended Finite Automata, in Proc. of ACM SIGCOMM, 2008.

[10] M. Becchi and S. Cadambi, Memory-Efficient Regular Expression Search Using State Merging, in Proc. of INFOCOM 2007.

[11] S. Kumar, B. Chandrasekaran, J. Turner and G. Varghese, Curing Regular Expressions Matching Algorithms from Insomnia, Amnesia, and Acalculia, in Proc. of ACM/IEEE ANCS, 2007.

[12] M. Becchi and P. Crowley, A Hybrid Finite Automaton for Practical Deep Packet Inspection, in Proc. of CoNEXT, 2007.

[13] B. L. Hutchings, R. Franklin and D. Carver, Assisting Network Intrusion Detection with Reconfigurable Hardware, in Proc. of the 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2002.

[14] F. Yu, R. H. Katz and T. V. Lakshman, Gigabit Rate Packet Pattern-Matching Using TCAM, in Proc. of the 12th IEEE International Conference on Network Protocols (ICNP), 2004.

[15] Y. Qi, K. Wang, J. Fong, Y. Xue, J. Li, W. Jiang and V. Prasanna, Feacan: Front-End Acceleration for Content-Aware Network Processing, in Proc. of INFOCOM, 2011.

[16] J. H. Holland, Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. U Michigan Press, 1975.

[17] D. E. Goldberg and K. Deb, A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. Urbana, 51 (1991), 61801-2996.

[18] G. Rudolph, Convergence Analysis of Canonical Genetic Algorithms. Neural Networks, IEEE Transactions on, 5(1) (1994), 96-101.

[19] A. Colorni, M. Dorigo and V. Maniezzo, Distributed Optimization by Ant Colonies, in Proc. of the first European conference on artificial life (Vol. 142, pp. 134-142).

[20] Regular Expression Processor, http://regex.wustl.edu/.