Computers and Electrical Engineering 67 (2018) 223–234

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Intelligent and efficient grouping algorithms for large-scale regular expressions[☆]

Zhe Fu[*,a,b], Kai Wang[d], Liangwei Cai[e], Jun Li[b,c]

[a] Department of Automation, Tsinghua University, Beijing, China
[b] Research Institute of Information Technology (RIIT), Tsinghua University, Beijing, China
[c] Tsinghua National Laboratory for Information Science and Technology (TNList), Beijing, China
[d] Yunshan Networks, Beijing, China
[e] College of Information Engineering, Shenzhen University, Shenzhen, Guangdong, China

ARTICLE INFO

Keywords:
Regular expression matching
Grouping algorithms
DFA
Intelligent optimization

ABSTRACT

Regular expressions are widely used in various applications. Due to its low time complexity and stable performance, Deterministic Finite Automaton (DFA) has become the first choice to perform fast regular expression matching. Unfortunately, compiling a large set of regular expressions into one DFA often leads to the well-known state explosion problem, and consequently requires huge memory consumption. Regular expression grouping is a practical approach to mitigate this problem. However, existing grouping algorithms are either brute-force or locally optimal and thus not efficient for large-scale regular expressions. In this paper, we propose two grouping algorithms, namely Reevo and Reant, to solve this problem intelligently and efficiently. In addition, two optimization methods are presented to accelerate the convergence speed and reduce the running time of proposed algorithms. Experimental results on large-scale rulesets from real world show that our algorithms achieve around 25% to 45% improvement compared to previous regular expression grouping algorithms.

1. Introduction

Nowadays, pattern matching is playing important roles in many practical applications. In network security, the payloads of network packets are scanned against a set of predefined patterns to identify the potential security threats, including viruses, intrusions, spams, data leakage and so forth. Bioinformatic research utilizes pattern matching to search for the similarity of DNA/RNA sequences. As the patterns are getting more and more complex, exact strings are being substituted gradually by regular expressions [1], which have more expressiveness and flexibility. To perform regular expression matching, regular expressions are compiled into Nondeterministic Finite Automaton (NFA) or Deterministic Finite Automaton (DFA). NFA has fewer states and transitions, and its space cost linearly depends on the size of regular expression set. Thus NFA is space-efficient. However, in the worst case, there can be massive state traversals per input character for NFA, leading to tremendous memory bandwidth requirement and slow matching speed. On the contrary, DFA only activates one state and requires precisely one state traversal per input character. As a result, DFA is fast, and thus it is the preferred choice for time-sensitive applications such as deep inspection in network security.

However, the time-efficiency of DFA is at the cost of the huge amount of memory consumption. Compared to NFA, DFA needs far more states and transitions, and suffers from the *state explosion* problem in many cases. What's more, the number of regular

☆ Reviews processed and recommended for publication to the Editor-in-Chief by Associate Editor Dr. Y. Sang.
* Corresponding author.
E-mail addresses: fu-z13@mails.tsinghua.edu.cn (Z. Fu), wangkai@yunshan.net.cn (K. Wang), cailw@szu.edu.cn (L. Cai), junl@tsinghua.edu.cn (J. Li).

https://doi.org/10.1016/j.compeleceng.2018.03.040
Received 30 June 2017; Received in revised form 23 March 2018; Accepted 23 March 2018
Available online 30 March 2018

expressions in practical use is increasing rapidly. The open-source network intrusion prevention system Snort [2] has already employed more than 27,000 rules[1] that are written in regular expressions. Such a large scale of regular expressions make it impossible to build a single DFA from these regular expressions on commodity processing platforms. Yu et al. [3] summarized several categories of regular expressions that can cause blow-up of corresponding DFA, and one major factor is concluded as the interactions among regular expressions. In general, for the regular expressions with interactions, the DFA constructed from them has a far larger size than the sum of the sizes of DFAs which are built from each regular expression individually. Based on this analysis, a number of regular expression grouping algorithms have been put forward to address this problem. However, the most state-of-the-art methods [3–6] can hardly find the optimum solution, and the techniques they employ do not fit for large-scale regular expressions.

In this paper, we propose novel grouping algorithms for regular expressions to solve the *state explosion* problem and reduce the memory consumption when regular expressions are constructed into DFAs. In detail, our contributions can be concluded as follows. First, we analyze the practical demands of regular expression grouping and formulate the grouping problem into two separate aspects: (1) when the number of groups is specified, minimize the total number of DFA states and (2) when the memory space of each group is limited, minimize the number of groups. Then, two intelligent and efficient grouping algorithms, namely Reevo and Reant, are proposed for each goal. In addition, we put forward two optimizations, including an improved approach to approximating the number of DFA states and a method to obtain better initial solutions, in order to accelerate the convergence speed and reduce the running time of the proposed algorithms. Experimental results based on practical large-scale rulesets show that our grouping algorithms could obtain around 25%–45% improvement compared to previous grouping algorithms.

This paper is organized as follows. Section 2 states the related work about regular expression matching. In Section 3, we illustrate the motivation of regular expression grouping and categorize the grouping problem into two kinds of situations based on practical demands. Two grouping algorithms are proposed correspondingly in Section 4. In Section 5, further optimizations for our grouping algorithms are put forward. Experimental results are shown in Section 6. In the last section, we give our conclusion.

2. Related work

During the past decades, many techniques for improving regular expression matching have been proposed. Most studies are aimed at compressing the transitions or states of DFA to reduce the memory consumption. Kumar et al. [7] invented D^2FA (Delayed input DFA) to reduce the number 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^2FA-derived algorithms (like A-DFA [8] *etc.*) are all designed for further improving the matching speed or compression ratio. PaCC [9] partitions a complex regular expression into multiple simple segments without semantic overlap, and uses a Relation Mapping Table (RMT) to record their dependencies. But this method only works for several types of complex regular expressions. Other techniques for reducing DFAs' space consumption such as state merging [10], character set reduction [11] similarly reduce memory at the cost of more matching time and worse temporal efficiency. Our methods are orthogonal and complementary to these methods.

Several studies leverage hardware platform to improve the executing efficiency of regular expression matching. FPGA based methods [12,13] 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 rulesets. GPU based methods [14] take advantage of the massive parallel execution units and high memory bandwidth, while the performance is sensitive to the divergences in memory accesses and execution path. TCAM devices are fast [15], but the updating algorithm may cost large amounts of storage space. Worse more, the high cost and big power consumption make TCAM devices not cost-effective for large-scale regular expression matching. The regular expression grouping methods can help obtain more efficient data layout on these hardware platforms to benefit maximally from hardware acceleration.

Regular expression grouping is a relatively new and independent direction to solve the *state explosion* problem of DFA. In [3], Yu et al. first put forward the ideas of greedily grouping regular expressions to construct multiple automata, and implemented a grouping algorithm that only uses the information whether a regular expression interacts with each other, without quantifying the interaction relationship. Moreover, the simplicity of this algorithm makes it run into local optimum easily. Becchi et al. [16] simply divided the rulesets binarily until the DFA states number of each group is smaller than a given limit. Rohrer et al. [4] evaluated the *distance* between each pair of regular expressions and converted the grouping problem to the Maximum Cut problem. Methods including Poles Heuristic and Simulated Annealing are leveraged based on the *distance* relationships among regular expressions. RegexGrouper [5] estimates the DFA size in a similar way by measuring the convolvement relationship. The authors map the k-grouping problem to the maximum k-cut problem in theory, and prove the grouping solution is not less than $1 - k^{-1}$ times of the optimal partition of the corresponding maximum k-cut problem. Other proposals are based on these ideas and get similar results [6]. However, all of these algorithms only find sub-optimal solutions within a reasonable running time, and could hardly deal with various situations where regular expressions are grouped.

[1] Extracted from snortrules-snapshot-29110, which was released on January 4, 2018.

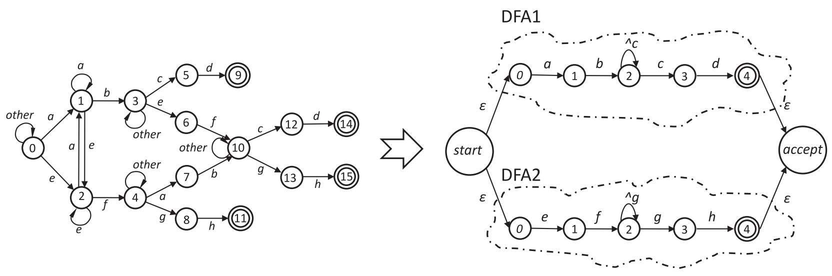


Fig. 1. Composing two DFAs by regular expression grouping.

3. Motivation and problem formulation

3.1. Regular expression grouping

Selectively distributing a set of regular expressions into several groups and constructing DFAs separately can reduce total memory usage, because the interactions among regular expressions can be isolated by multiple DFAs. Fig. 1 shows a simple example which distributes two regular expressions ("ab.*cd" and "ef.*gh") into two groups. When compiling these two regular expressions into one DFA, it has 16 DFA states. After grouping, the total number of DFA states decreases from 16 to 10. Since the memory cost by DFAs is proportional to the number of states in DFAs, the reduction in states number equals to the decrease of memory consumption. By adding two new states: a start state and an accept state, and four ε transitions that link individual DFAs to the newly added states, the two separate DFAs perform the equivalent matching function as the composite one.

Different grouping solutions of regular expressions have different processing complexity and different storage cost. Suppose there are m regular expressions with average length n, there are at least two approaches: one is to compile each regular expression individually into m DFAs, and the other is to compile them altogether into a single DFA. The processing complexity for the former method is $O(m)$, and the storage cost is $O(m \times |\Sigma|^n)$. For the latter method, the processing complexity is exactly $O(1)$ while the storage cost surges to $O(|\Sigma|^{n \times m})$. Regular expression grouping, however, can be regarded as a trade-off between these two approaches. If m regular expressions are distributed into k groups ($1 \le k \le m$), the processing complexity is distinctly $O(k)$, and the storage cost becomes $O(k \times |\Sigma|^{n \times m/k})$.

Although the huge space cost caused by the DFA *state explosion* problem can be effectively mitigated via regular expression grouping, it doesn't make any sense to focus only on the total number of DFA states or the number of groups which regular expressions are distributed into. On one hand, minimizing only the total number of DFA states will end up with a large group number, which will increase the processing complexity. On the other hand, minimizing only the number of groups will apparently get back to DFA *state explosion*. In other words, we should obtain a better trade-off between processing complexity and storage cost according to different demands of various application scenarios.

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

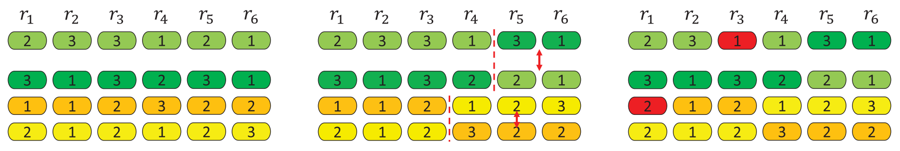
In practical use, the group number of regular expressions is limited by the maximum number of DFAs a particular hardware platform supports to execute simultaneously. If the number of groups is smaller than that of cores/threads, several cores/threads are in the idle state, which fails to make full use of the computational resource. Conversely, if more DFAs are generated than the cores/threads, the excess DFAs will not provide any performance improvement, but bring more overhead and latency. Therefore, for the hardware platform with k cores/threads available or allocated for regular expression matching, the optimal number of groups (DFAs) is k.

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

Problem 1. For a regular expression set $\mathscr{S} = \{r_1, r_2, ..., r_i, ...\}$, $1 \le i \le m$ and a given k, $1 \le k \le m$, the goal is to find k disjoint subsets $\mathscr{R}_1, \mathscr{R}_2, ..., \mathscr{R}_k$, $\bigcup_{j=1}^{k} \mathscr{R}_j = \mathscr{S}$, to minimize the overall DFA states number $\sum_{j=1}^{k} T(\mathscr{R}_j)$, where $T(\mathscr{R}_j) = T\left(\sum_{r_i \in \mathscr{R}_j} r_i\right)$ denotes the number of states of the DFA constructed by subset $\mathscr{R}_j$.

3.3. Given upper limit of DFA size, minimize the number of groups

In some application scenarios, the memory cost by DFAs is limited by the physically available memory size of a particular hardware platform (such as certain embedded devices), which results in the fact that the maximum memory usage, rather than the



(a) Initialization (b) Crossover (c) Mutation

Fig. 2. Example of Reevo algorithm for distributing 6 regular expressions into 3 groups.

number of groups, becomes the primary goal for implementing regular expression matching on these platforms. For example, Xilinx Zynq-7000 series FPGAs have up to 4.9Mb Block RAM[2], which is still too small for large-scale regular expression rulesets. Therefore, if the regular expression rulesets could be distributed into as few groups as possible within the limit of memory space, we can utilize least hardware resources to process all DFAs, and meanwhile reduce the latency of throughput and accelerate the matching speed. The target of the problem in this case can be mathematically defined as follows:

Problem 2. For a regular expression set $\mathscr{S} = \{r_1, r_2, \ldots, r_i, \ldots\}$, $1 \le i \le m$ and a given upper limit THR, the aim is to find k disjoint subsets $\mathscr{R}_1, \mathscr{R}_2, \ldots, \mathscr{R}_k$, $\bigcup_{j=1}^{k} \mathscr{R}_j = \mathscr{S}$, $1 \le k \le m$, where k is minimal, and $\forall j \in [1, k]$, there exists $T(\mathscr{R}_j) \le THR$, where $T(\mathscr{R}_j) = T\left(\sum_{r_i \in \mathscr{R}_j} r_i \right)$ denotes the number of states of the DFA constructed by subset $\mathscr{R}_j$.

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

4. Grouping algorithms

4.1. Why use intelligent algorithms

From the analysis above, the regular expression grouping problem can be abstracted into optimization problems. An accurate optimum solution can be obtained by traversing through all possible solutions for a small-scale optimization problem. However, for a set of m regular expressions which is to be distributed into k groups, there exists $(\prod_m k)/k! = k^m/k!$ distinct solutions. The exponential growth makes the number of possible distributions too enormous to find the optimal solution. Take a practical ruleset used in deep inspection for example. The ruleset of L7-filter [17] is relatively small, which is composed of 111 regular expressions. If distributing these regular expressions into two groups, the number of distinct possible distributions is $2^{111}/2! = 2^{110} \approx 1.3 \times 10^{33}$. For larger rulesets such as Snort [2], the grouping possibilities grow exponentially. This problem is proved to be NP-complete [5]. Thus, it is obviously infeasible to use a brute-force method to find the optimal or close-to-optimal grouping solution from all possible situations, and we need intelligent algorithms to find the optimum solutions to the regular expression grouping problem. Section 3 categorizes the grouping problem into two separate cases according to practical needs. Correspondingly, two intelligent grouping algorithms are proposed in this section, being called Reevo and Reant respectively, to handle each situation.

4.2. Reevo – minimize DFA states under given group number

When the group number k is specified, this grouping problem (Problem 1) is similar to the Bounded Knapsack Problem (BKP). Genetic Algorithm (GA) proves to be an efficient method to solve this type of problems. Inspired by GA, we propose Reevo algorithm to obtain efficient grouping solutions under given group number. However, the chromosomes of individuals in classical GA are represented in binary strings of 0s and 1s, yet this representation is not suitable for regular expression grouping problem. Thus, a novel index structure instead of the conventional binary chromosome is proposed to represent one grouping solution of regular expressions. For a problem of distributing a set of m regular expressions into k groups, each chromosome in Reevo represents one grouping solution of regular expressions. The length of all chromosomes is m, and each position in the chromosomes represents one corresponding regular expression, and the value (ranging from 1 to k) represents the group label which the regular expression is placed into.

Fig. 2a depicts a simple example of encoding 6 regular expressions into chromosomes when k is set to 3. As can be seen, each individual corresponds to one chromosome, and the length of the chromosome is determined by the number of regular expressions, and all values on the chromosome are between 1 and 3. The first digit on the first chromosome indicates that in this grouping solution, the regular expression r_1 is to be placed into group 2. By analogy, the first chromosome {2 3 3 1 2 1} encodes one grouping solution that the fourth, sixth regular expressions are assigned to Group 1, the first, fifth are assigned to Group 2, and the second, third are assigned to Group 3. In Fig. 2a, the other three chromosomes are encoded in the same manner as the initial solutions of Reevo.

To evaluate each chromosome, we define the unfitness in our algorithm instead of the fitness in classical GA. The unfitness of the chromosome is exactly the number of total DFA states. Mathematically, for the chromosome $\mathscr{C} = \{c_1\ c_2\ \ldots\ c_i\ \ldots\ c_m\}$, the unfitness $\mathscr{U}$ is

[2] Xilinx Zynq Z-7020 has 140 Blocks, and each block has 36Kb RAM.

calculated by $\mathscr{U}(\mathscr{C}) = \sum_{j=1}^{k} T\left(\sum_{r_i \in \mathscr{S}_j} r_i\right)$, where $\mathscr{S}_j$ denotes the set of regular expressions in Group j. $\forall i \in [1, m]$, if $c_i = j$, then $r_i \in \mathscr{S}_j$. After unfitness calculation, individuals are selected according to Roulette Wheel Selection [18]. Small unfitness of a chromosome often leads to a large possibility for the chromosome to be chosen for further evolution. In Reevo, for a population of l chromosomes $\mathscr{C}_1, \mathscr{C}_2, ..., \mathscr{C}_l$, the probability for $\mathscr{C}$ to be selected is calculated as: $1 - \mathscr{U}(\mathscr{C})/\sum_{i=1}^{l} \mathscr{U}(\mathscr{C}_i)$. Next, the chromosome performs crossover with the other one with a probability of p_c, and then a random digit in the chromosome is picked and mutates with a probability of p_m. p_c and p_m are two configurable parameters which determine the global optimality and convergence speed of Reevo.

As shown in Fig. 2b, the chromosomes of the first and second individuals are selected as an example, and then a position is chosen randomly (between the fourth and fifth digits in the figure) and the two chromosomes are crossovered with the probability of p_c. This operation is also performed for the remaining two chromosomes. After that, a digit on the chromosome is picked randomly and mutated into an arbitrary value between 1 and k with the probability of p_m. Based on the above steps, the Reevo algorithm generates a new population with four different individuals, as shown in Fig. 2c.

Using Finite Markov Chains, Rudolph et al. [19] prove that this type of algorithms with the process of crossover, mutation, and selection operations cannot guarantee the convergence to the global optimum. For this reason, Reevo adopts an Elitist Strategy which maintains the best solution found over time before next-round selection, *i.e.*, find the individual with the worst chromosome in the current generation and replace it with the best chromosome from the last generation. The global convergence is proved in theory when this strategy is taken [19].

The pseudo-code of Reevo is as shown in Algorithm 1. The operations of crossover and mutation (Line 8 - Line 11) guarantee that the algorithm will not fall into local optimum, meanwhile the Roulette Wheel Selection (Line 6 - Line 7) and Elitist Strategy (Line 4, Line 13 - Line 14) accelerates the convergence speed.

4.3. *Reant – minimize group number under given space limitation*

If the space of each group is limited, the goal becomes to obtain as few groups as possible. This problem (Problem 2) could be regarded as a new variant of partition problems, which are generally solved by greedily algorithms or dynamic programming. However, existing methods may be trapped in a local optimum easily. To achieve better solutions, we propose Reant algorithm, which takes advantage of the two core ideas from Ant Colony Optimization algorithm (ACO), namely probabilistic wandering and positive feedback.

Probabilistic wandering

In Reant, whether two or more regular expressions are distributed into the same group will be determined by the pre-estimated probability. With reference to the upper limitation *THR*, the smaller the space cost by DFAs is, the greater the probability being in the same group should be. Further, the probability function is designed to be non-linear so that the regular expressions which cause less *state explosion* will have much greater chances to be put together in the same group.

For a regular expression set $\mathscr{S} = \{r_1, r_2, ..., r_i, ...\}$, and a given space limitation *THR*, the probability of adding regular expression r_i into the subset $\mathscr{R}_m$ is: (pheromones omitted here)

$$\begin{aligned} p_{add}(\mathscr{R}_m, i) &= \frac{\log(THR) - \log\left(T\left(\sum_{r_k \in \mathscr{R}_m} r_k + r_i\right)\right)}{\log(THR)} \\ &= 1 - \frac{\log\left(T\left(\sum_{r_k \in \mathscr{R}_m} r_k + r_i\right)\right)}{\log(THR)} \end{aligned} \tag{1}$$

where $T(\mathscr{R}_m)$ represents the number of states of the DFA constructed by regular expression set $\mathscr{R}_m$. Since $T(\mathscr{R}_m)$ is always greater than 0, $p_{add}(\mathscr{R}_m, i)$ would be no larger than 1. On the other side, if $T\left(\sum_{r_k \in \mathscr{R}_m} r_k + r_i\right) > THR$, $p_{add}(\mathscr{R}_m, i)$ will below zero, which means that there is no possibility to put regular expressions r_i into the group $T(\mathscr{R}_m)$. Under normal circumstances, $p_{add}(R_m, i) \in (0, 1)$.

Positive feedback

We also employ the idea of pheromones from ACO algorithm. *pheromone*(*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 tried to be put 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 $\mathscr{R}_m$ is:

$$p'_{add}(\mathscr{R}_m, i) = (1 - \alpha)p_{add}(\mathscr{R}_m, i) + \alpha \frac{\sum_{r_k \in \mathscr{R}_m} pheromone(i, k)}{|\mathscr{R}_m|} \tag{2}$$

where α is the weight of pheromones, $|\mathscr{R}_m|$ stands for the number of regular expressions in group $\mathscr{R}_m$, and $\sum_{r_k \in \mathscr{R}_m} pheromone(i, k)/|\mathscr{R}_m|$ means the average amount of pheromones between regular expression r_i and the existing group $\mathscr{R}_m$. It is clear that more pheromones lead to a higher probability of adding r_i into $\mathscr{R}_m$. If r_i is added into the group $\mathscr{R}_m$, the pheromones between r_i and each regular expression of $\mathscr{R}_m$ increase as: $\Delta pheromone(i, k) = p_{add}(\mathscr{R}_m, i) \times \beta, \forall r_k \in \mathscr{R}_m$, where β is the weight of pheromones that are laid down. The pheromones evaporate by the parameter γ, *i.e.*, $pheromone(i, j) = pheromone(i, j) \times (1 - \gamma), \forall r_i, r_j \in \mathscr{S}$. The only thing to note here is that the maximum value of pheromones is 1,

```
$\mathcal{P} \leftarrow$ initialization();
**while** *termination criterion has not been met* **do**
    cal_unfitness($\mathcal{P}$);
    $C_{best} \leftarrow$ find_best($\mathcal{P}$);
    **while** *size_of($\mathcal{P}'$) <size_of($\mathcal{P}$)* **do**
        $C_i \leftarrow$ roulette_wheel_select($\mathcal{P}$);
        $C_j \leftarrow$ roulette_wheel_select($\mathcal{P}$);
        $C_i', C_j' \leftarrow$ crossover $C_i, C_j$ with probability $p_c$ ;
        $C_i' \leftarrow$ mutate $C_i'$ with probability $p_m$;
        $C_j' \leftarrow$ mutate $C_j'$ with probability $p_m$;
        add $C_i'$ and $C_j'$ to population $\mathcal{P}'$;
    **end**
    $C_{worst} \leftarrow$ find_worst($\mathcal{P}$);
    $C_{worst} \leftarrow C_{best}$;
    $\mathcal{P} \leftarrow \mathcal{P}'$;
    $\mathcal{P}' \leftarrow \emptyset$;
**end**
**return** find_best($\mathcal{P}$);
```

Algorithm 1. Reevo – minimize DFA states under given group number.

```
1  while termination criterion has not been met do
2      put_all_ants_in_initial_state();
3      for each $ant_i$ in the colony do
4          $cnt_i = 0$;
5          add a random $re$ to $ant_i$ as $group[cnt_i]$;
6      end
7      while all ants have space to go do
8          for each $ant_i$ in the colony do
9              for each $re_j$ remained for $ant_i$ do
10                 add $re_j$ to $ant_i$ with probability $P_{add}\big(group[cnt_i], re_j\big)$ ;
11                 if $re_j$ added to $group[cnt_i]$ then
12                     pheromones_increase($re_j, group[cnt_i]$);
13                 end
14             end
15             if no $re$ added to $group[cnt_i]$ then
16                 $cnt_i = cnt_i + 1$;
17                 add a random $re$ to $ant_i$ as $group[cnt_i]$;
18                 go to Line 9;
19             end
20         end
21     end
22     $result \leftarrow$ best ant;
23     pheromones_evaporate();
24 end
25 return best of $results$;
```

Algorithm 2. Reant – minimize number of groups under given space limitation.

otherwise, the pheromones will increase uncontrollably even if there exists a process of evaporation.

The pseudo-code of Reant is as shown in Algorithm 2. “Ants” are used to simulate the process of regular expression grouping. The merit of grouping regular expressions with probabilities (Line 10) lies in the avoidance of converging to a locally optimal solution, and the introduction of pheromones (Line 11 - Line 13) facilitates the positive feedback and accelerates the convergence speed. In [20], the authors prove the convergence of this class of algorithms.

5. Optimizations

Due to the fact that both Reevo and Reant algorithms require a lot of iterative computations, calculating the accurate number of total states by building DFA in each-round iteration is impractical. Through testing, for the problem of distributing 111 regular expressions into 4 groups with given 50 individuals and 5000 iterations in Reevo, it takes nearly 20 h to obtain a final solution by using the Regular Expression Processor [21], and the time cost may grow exponentially with the increase of regular expressions and groups that regular expressions are distributed into. What’s worse, the randomness of both Reevo and Reant may lead to a situation in which the construction of a temporal DFA is infeasible. In this section, two optimization methods are proposed to accelerate the convergence of Reevo and Reant algorithms.

5.1. Fast DFA states calculating

The convolvement relationship of regular expressions is proposed in [5] to estimate DFA states. First, the coefficient a_i is defined as the number of states of the DFA constructed by regular expression r_i itself, and the coefficient $b_{i,j}$ is defined as the number of states of the DFA constructed by regular expression pair r_i and r_j. Then for a regular expression ruleset $\mathscr{S}$, the number of DFA states can be estimated by the following formula:

$$E(\mathscr{S}) = \sum_{r_i \in \mathscr{S}} a_i + \sum_{r_i, r_j \in \mathscr{S}, i<j} (b_{i,j} - a_i - a_j) \tag{3}$$

The experimental results (shown in Section 6.1) point out that although Formula (3) can be an applicable approximation on the condition of a small quantity of regular expressions, the increase of accurate number of DFA states is far greater than the approximate number for large rulesets, which will cause remarkable bias for the optimization goals. To achieve better approximations, we define another coefficient $\rho_{i,j}$ as the expansion rate of regular r_i and r_j: $\rho_{i,j} = (b_{i,j} - a_i - a_j)/(a_i + a_j) = w_{i,j}/(a_i + a_j)$. The increase of the approximate number of states is relevant to not only $w_{i,j}$, but also the expansion rate $\rho_{i,j}$, especially in the situation where there exists a relatively huge number of regular expressions or serious *state explosion*. When adding a third regular expression r_l into the group $\mathscr{R}_m$, according to the improved approximation, the increase in the number of states is:

$$\Delta E = \sum_{r_i \in \mathscr{R}_m} \left(\sum_{r_j \in \mathscr{R}_m, i \neq j} (\rho_{l,j} + \rho_{i,j}) \times w_{l,i} \right) \tag{4}$$

Then the sum of states of group $\mathscr{R}_m$ when compiled into the same DFA is:

$$E(\mathscr{R}_m) = \sum_{r_i \in \mathscr{R}_m} a_i + \sum_{r_{i,j} \in \mathscr{R}_m, i<j} \; \sum_{r_l \in \mathscr{R}_m, l \neq i,j} (\rho_{i,l} + \rho_{l,j}) \times w_{i,j} \tag{5}$$

For a regular expression set $\mathscr{S} = \{r_1, r_2, ..., r_i, ...\}$ which is to be distributed into k groups, the new DFA states approximation $E(S) = \sum_{m=1}^{k} E(\mathscr{R}_m)$. In Section 6.1, a comparison experiment between the previous method and our improvement is conducted. The result shows that the improved method which uses the additional coefficient $\rho_{i,j}$ results in far better approximation in both numeric and variation tendency.

5.2. Initial solution set improving

Another optimization for fast convergence is to improve the initial solution set of both Reevo and Reant algorithms. Rather than random solutions, a better initial solution will avoid unnecessary iterations and reduce algorithms running time. In Section 5.1, $w_{i,j} = b_{i,j} - a_i - a_j$ is denoted as a coefficient to quantify the relationship between regular expression r_i and r_j. Larger $w_{i,j}$ implies that the DFA states will expand more if r_i and r_j are grouped together. For a regular expression set $\mathscr{S} = \{r_1, r_2, ..., r_i, ...\}$, $1 \le i \le m$, which is to be distributed into k groups, a graph that consists of m nodes and $m \times (m-1)/2$ edges could be built, and the weight of each edge is $w_{i,j}$. Taking advantage of the ideas from spectral clustering [22], we can obtain a quick initial grouping solution that is far better than the random one. The steps are as follows.

Step 1: Generate the affinity matrix $A \in \mathbb{R}^{m \times m}$ by $A_{i,j} = \exp(-w_{i,j}/\sigma)$ when where σ is a free parameter. If $i = j$, $A_{i,j} = 0$.
Step 2: Calculate the unnormalized graph Laplacian matrix L by $L = D - W$, where D is a degree matrix (D’s each diagonal element is the sum of A’s each row).
Step 3: Find k smallest eigenvalues from matrix L, and form a new matrix $X \in \mathbb{R}^{m \times k}$ by stacking the corresponding k eigenvectors.
Step 4: Treat each row of X as a vector in k-dimensional space, and use *K-means* to cluster into k groups.

Table 1
Rulesets used in the evaluations.

name	ruleset01	ruleset02	ruleset03	ruleset04	ruleset05	ruleset06
# of regexes	111	120	1190	1196	1386	5886
source	L7-filter [17]	Snort [2]	Snort [2]	Bro [23]	commercial	commercial
state explosion	moderate	serious	serious	serious	serious	moderate

Experiments in Section 6.4 show that this method could obtain a satisfactory initial solution in a short time.

6. Evaluations

In this section, we conduct a series of experiments to evaluate the optimality and efficiency of our intelligent grouping algorithms. Six rulesets picked from the open-source software and commercial companies are used, as shown in Table 1. Ruleset01 comes from L7-filter [17], a popular application layer packet classifier for Linux. Ruleset02, ruleset03 and ruleset04 are from network intrusion detection and prevention systems Snort [2] and Bro [23]. We also obtain two large rulesets from a major commercial networking vendor. Experiments are conducted on an Intel(R) Core(R) i7-4790 platform with 4 × 64 KB L1 Cache, 4 × 256 KB L2 Cache, 8 MB L3 Cache, 8 GB memory and Ubuntu 14.04 operating system. Our grouping algorithms are implemented in C+ +. The Regular Expression Processor [21] is used to calculate the number of DFA states of each regular expression grouping solution.

6.1. DFA states approximation

In Section 5.1, a fast DFA states approximation method is proposed. Among all the existing grouping algorithms, [5] is the only one that presented a similar approximation method to estimate DFA states before grouping. It is infeasible to compare every accurate DFA states number value with the approximations in all possible grouping solutions. Therefore, we repeatedly add one random regular expression to a test group and calculate the states number of the DFA constructed from this test group. Fig. 3 depicts the comparison of the accurate states number, DFAestimator [5] and our improved approximation when the number of regular expressions in the test group is increasing. It is shown that if the test group contains only a fewer regular expressions (fewer than 10), both of the previous method and our improved approximation are close to the accurate number. However, if more regular expressions are added to the test group, our improved method and the accurate value increase following the same trend, while the approximation of DFAestimator increases much slowly. When 20 or more regular expressions are contained in the test group, the DFA states number of our improved approximation and the accurate value are around 10 times as huge as that of the previous method. When more than 24 regular expressions are added to the test group, the space consumption of the corresponding DFA is beyond the memory limit of our experimental machine.

Table 2 shows more evaluation results. A number of regular expression subsets of size 1, 2, 4, 8, 12 and 16 are randomly picked from the original rulesets, and the average deviations between the approximations and accurate states are recorded after multiple experiments. It is observed that for large subsets, our method is much more accurate than DFAestimator. On average, the deviation of our method is around 11%, while DFAestimator gets a deviation of about 43%. On the other side, the computational overhead of our improved method is negligible since it only requires extra addition and multiplication operations compared to the previous work. As one can see from the figure and the table, in both numeric and tendency, our improved method which adds the coefficient $\rho_{i,j}$ leads to far better approximation than existing work, which brings more practical grouping solutions to regular expressions.

6.2. Grouping results of Reevo

This subsection shows the experimental results when the regular expression group number is specified. We implement three algorithms from state-of-the-art work as comparisons: Pole Heuristic and Simulated Annealing from Rohrer et al. [4], and Regex-Grouper from Liu et al. [5]. These methods also group a regular expression ruleset into a specified number of subsets. For Reevo, p_c is set to 0.7, p_m is set to 0.1, the number of individuals is 20, and the maximum number of iterations is 2500. In Fig. 4a and b, the number of regular expression groups is specified from 2 to 8. From these figures we observe that Reevo achieves the best grouping results among these four algorithms. For example, when distributing ruleset01 into 3 subsets, the total DFA states of Reevo is only

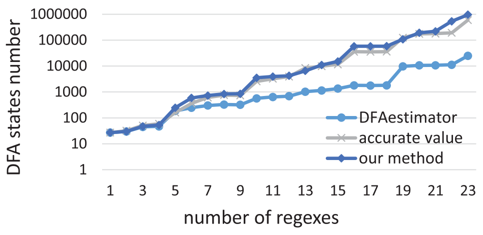


Fig. 3. Comparisons of different DFA states approximation methods.

Table 2
Average deviation of different approximation methods on subsets with different sizes.

size of regexes	1	2	4	8	12	16
DFAestimator	0.00%	6.25%	20.33%	56.61%	82.49%	94.96%
Our method	0.00%	6.25%	10.16%	14.44%	7.52%	30.55%

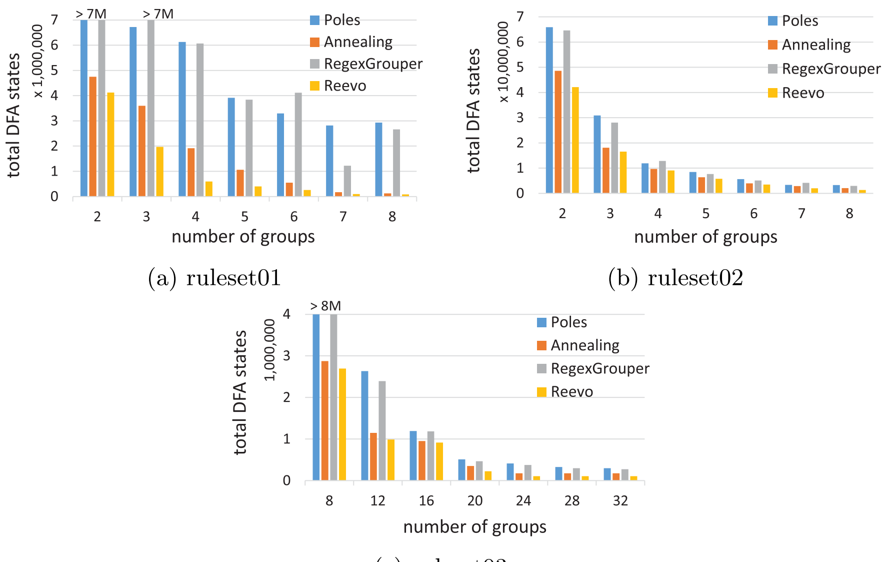


Fig. 4. Comparison of total DFA states on different rulesets under given group number.

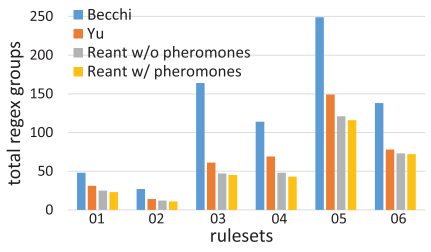
1.97 million, while Simulated Annealing obtains 3.59 million, Pole Heuristic obtains 6.72 million and RegexGrouper gets more than 7 million DFA states. Pole Heuristic and RegexGrouper adopt similar heuristic algorithms, so there is no significant difference between them. What is more, the results of them are unstable, which can be ascribed to the local optimum that they trap into easily. Simulated Annealing is also an iterative algorithm, but it ends up with results that are 10%–200% worse than Reevo. Large-scale ruleset experiments (shown in Fig. 4c) demonstrate the similar effectiveness of the presented algorithm. After making statistics and analysis of multiple grouping results on different rulesets, it is concluded that Reevo saves 15% at least and 45% on average memory consumption compared to other regular expression grouping algorithms, since Reevo can find an optimal solution in each situation, while others can only achieve sub-optimal results.

6.3. Grouping results of Reant

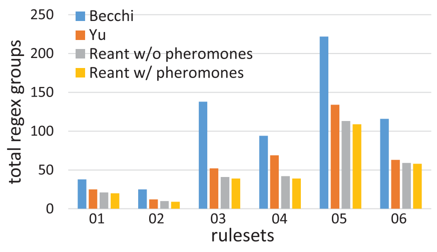
To evaluate grouping methods when the space of each group is limited, we compare our Reant algorithm with Becchi's approach [16] and Yu's algorithm [3], which also distribute regular expressions under given space limitations of each group. In this situation, the aim is to find a grouping solution which generates as few groups (DFAs) as possible. For Reant, the parameter α is set to 0.5, β is set to 0.7, γ is set to 0.3, the number of "ants" is 20, and the maximum number of iterations is 50. To verify the effect of pheromones, we also run Reant algorithm with and without the pheromones. Fig. 5 shows the experimental results under different conditions. When the state limit is set to 5000 (Fig. 5a), Becchi's approach gets the worst results since it divides the rulesets in binary until the number of DFA states is smaller than the limitation. Yu's algorithm adopts heuristic information and obtains better results than Becchi's approach. However, the insufficient information in Yu's algorithm makes it only achieve the local optimum. Reant algorithm generates obviously fewer groups than Yu's algorithm, and with the same parameters, the results of Reant with pheromones are better than that without pheromones. For example, when distributing ruleset02 into groups with the space limitation of 5000 states, Becchi's approach generates 27 groups, Yu's algorithm generates 14 groups, and Reant only produces 12 groups without pheromones and 11 groups with pheromones. Experiments under other conditions (Fig. 5b and c) show the same results. For all rulesets, Reant achieves fewer groups in each situation by acquiring more optimal grouping solutions of regular expressions compared to existing methods. Specifically, Reant reduces around 55% groups number on average than Becchi's approach and around 25% groups number on average than Yu's algorithm.

6.4. Initial solution and grouping efficiency

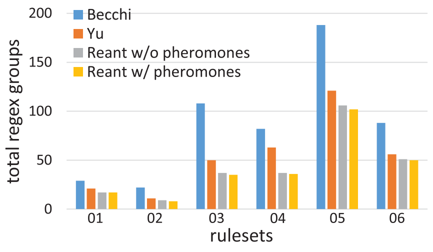
This subsection shows the benefits of the optimized initial solutions introduced in Section 5.2. Fig. 6 depicts the comparison of



(a) THR=5000



(b) THR=10000



(c) THR=20000

Fig. 5. Comparison of total regex groups on different rulesets and different space limitations.

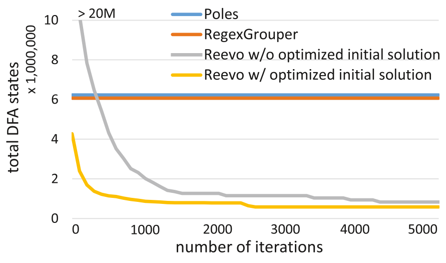


Fig. 6. Convergence of RegexGrouper and Reevo on ruleset01.

Table 3
Time cost of our improvement initial solution and existing methods.

Number of groups	2	3	4	5	6	7	8
Pole Heuristic (ms)	15.1	22.5	25.4	32.6	34.5	37.8	38.3
DFAgrouper (ms)	18.1	30.9	30.2	39.0	43.7	39.2	50.8
Our initial solution (ms)	15.7	17.2	19.5	20.4	22.0	22.3	24.1

Pole Heuristic, Simulated Annealing, RegexGrouper and Reevo with/without this optimization. In this case, regular expressions in ruleset01 are grouped into 4 groups. Since Pole Heuristic and RegexGrouper are not iterative algorithms, their grouping results are constant: the number of total DFA states obtained by Pole Heuristic is about 6.23 million and that by RegexGrouper is about 6.07 million. The result of Reevo with optimized initial solution converges to about 0.58 million after 2500 iterations, while Reevo without optimized initial solution only obtains a final grouping solution of 0.82 million states after 5000 iterations. Obviously, the optimized initial solution speeds up the convergence of our proposed algorithms.

It must be noted that both Reevo and Reant spend more running time compared with existing grouping algorithms. However, the grouping algorithms often run offline so their consuming time is not a major concern in the situation where regular expression rules are not frequently updated. To say the least, even if the running time of grouping algorithms is strictly limited, the optimized initial solution which could be obtained in a very short time is acceptable (even faster and better than existing grouping methods). Table 3 shows the time consumption of our improvement initial solution and non-iterative algorithms on ruleset01. Our optimized initial solution requires about 30%–50% less time and achieves better grouping results than existing methods.

7. Conclusion

To reduce the huge memory consumption when performing regular expression matching on large-scale rulesets, we propose intelligent and efficient regular expression grouping algorithms. The practical demands of regular expression grouping are analyzed and the grouping problem is formulated into two aspects. Two grouping algorithms are proposed accordingly and obtain better results than state-of-the-art grouping methods by acquiring the global optimum solution of regular expression distributions. When the number of groups is given, our proposed algorithm achieves 45% less memory consumption than previous work on average. Under the circumstance of limited memory space for each group, our proposed algorithm generates around 25% fewer regular expression groups in comparison with other methods, and hence requires fewer hardware resources for regular expression matching. We also present an improved DFA states calculation method which achieves far better DFA size approximation when the ruleset is large. The conducted evaluation shows that our approximation method reduces about 30% deviation compared to existed methods. In addition, a new initial grouping solution generation approach is proposed, which obtains better initial solutions in a very short time and accelerates the convergence speed of our algorithms. Our future work will focus on the fast incremental grouping of updated regular expressions.

References

[1] Xu C, Chen S, Su J, Yiu S, Hui LC. A survey on regular expression matching for deep packet inspection: applications, algorithms, and hardware platforms. IEEE Commun Surv Tutorials 2016;18(4):2991–3029.
[2] Snort. https://www.snort.org.
[3] Yu F, Chen Z, Diao Y, Lakshman T, Katz RH. Fast and memory-efficient regular expression matching for deep packet inspection. Proceedings of the 2006 ACM/IEEE symposium on architecture for networking and communications systems. ACM; 2006. p. 93–102.
[4] Rohrer J, Atasu K, van Lunteren J, Hagleitner C. Memory-efficient distribution of regular expressions for fast deep packet inspection. Proceedings of the 7th IEEE/ACM international conference on hardware/software codesign and system synthesis. ACM; 2009. p. 147–54.
[5] Liu T, Liu AX, Shi J, Sun Y, Guo L. Towards fast and optimal grouping of regular expressions via dfa size estimation. IEEE J Sel Areas Commun 2014;32(10):1797–809.
[6] Liangwei C, Haoping Y. Regular expression grouping optimization based on shuffled frog leaping algorithm. Computer and communications (ICCC), 2016 2nd IEEE international conference on. IEEE; 2016. p. 1111–5.
[7] Kumar S, Dharmapurikar S, Yu F, Crowley P, Turner J. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. ACM SIGCOMM computer communication review. 36. ACM; 2006. p. 339–50.
[8] Becchi M, Crowley P. A-dfa: A time-and space-efficient dfa compression algorithm for fast regular expression evaluation. ACM Trans Architect Code Optim (TACO) 2013;10(1):4.
[9] Wang K, Fu Z, Hu X, Li J. Practical regular expression matching free of scalability and performance barriers. Comput Commun 2014;54:97–119.
[10] Liu AX, Torng E, Liu AX, Torng E. Overlay automata and algorithms for fast and scalable regular expression matching. IEEE/ACM Trans Netw (TON) 2016;24(4):2400–15.
[11] Tang Q, Jiang L, Dai Q, Su M, Xie H, Fang B. Rics-dfa: a space and time-efficient signature matching algorithm with reduced input character set. Concurrency Comput 2017;29(20).
[12] Chen X, Jones B, Becchi M, Wolf T. Picking pesky parameters: optimizing regular expression matching in practice. IEEE Trans Parallel Distrib Syst 2016;27(5):1430–42.
[13] Qi Y, Wang K, Fong J, Xue Y, Li J, Jiang W, et al. Feacan: front-end acceleration for content-aware network processing. INFOCOM, 2011 proceedings IEEE. IEEE; 2011. p. 2114–22.
[14] Yu X, Becchi M. Exploring different automata representations for efficient regular expression matching on gpus. ACM SIGPLAN notices. 48. ACM; 2013. p. 287–8.
[15] Meiners CR, Patel J, Norige E, Liu AX, Torng E. Fast regular expression matching using small tcam. IEEE/ACM Trans Netw (TON) 2014;22(1):94–109.
[16] Becchi M, Franklin M, Crowley P. A workload for evaluating deep packet inspection architectures. Workload characterization, 2008. IISWC 2008. IEEE international symposium on. IEEE; 2008. p. 79–89.
[17] L7-filter. http://l7-filter.sourceforge.net.
[18] Goldberg DE, Deb K. A comparative analysis of selection schemes used in genetic algorithms. Found Genet Algorithms 1991;1:69–93.
[19] Rudolph G. Convergence analysis of canonical genetic algorithms. IEEE Trans Neural Netw 1994;5(1):96–101.
[20] Stutzle T, Dorigo M. A short convergence proof for a class of ant colony optimization algorithms. IEEE Trans Evol Comput 2002;6(4):358–65.
[21] Regular expression processor. http://regex.wustl.edu.
[22] Ng AY, Jordan MI, Weiss Y. On spectral clustering: analysis and an algorithm. NIPS. 14. 2001. p. 849–56.
[23] Bro. https://www.bro.org.

Zhe Fu is currently a Ph.D. student at Tsinghua University, China. He received the B.S. degree in the Department of Automation from Tsinghua University, in 2013. His research interests focus on pattern matching and traffic management.

Kai Wang works at Yunshan Networks, Beijing, China. He received the Ph.D. degree in the Department of Automation from Tsinghua University, in 2014. His research interests focus on high performance regular expression matching.

Liangwei Cai is currently Professor of Shenzhen University, China. He received the M.S. and B.S. degrees in Automation from Tsinghua University. His research interests focus on intelligent optimizations.

Jun Li is currently Professor of Research Institute of Information Technology, Tsinghua University. He holds a Ph.D. degree in CS from New Jersey Institute of Technology, and M.S. and B.S. degrees in Automation from Tsinghua University. His research interest is in network security and software-defined networking.