第44卷第1期 中国科学技术大学学报 Vol. 44, No. 1
2014年1月 JOURNAL OF UNIVERSITY OF SCIENCE AND TECHNOLOGY OF CHINA Jan. 2014

文章编号:0253-2778(2014)01-0001-11

基于状态子集编码的快速DFA构造算法

彭坤杨

(中国科学技术大学计算机科学与技术学院,安徽合肥 230027)

摘要:网络深度包检测等网络应用广泛采用正则表达式匹配技术检测网络中的传输内容,正则表达式用非确定性有限自动机(NFA)或者确定性有限自动机(DFA)实现.网络应用对匹配速度要求很高,相比NFA,DFA具有确定性的匹配速度,但所有基于DFA的方法需要预先从NFA构造一个与之等价的DFA,于是DFA的构造成为系统瓶颈之一.为此通过深入探索自动机内在运行特性——NFA状态间活跃关系和NFA中导致DFA空间膨胀的因素,设计了一种NFA状态子集的编码方法和查询方法,显著减少了DFA构造过程中状态子集的查询代价.基于入侵检测与防护系统Snort中的真实规则集的实验表明,与传统的子集构造算法相比,该方法减少了88.33%~93.57%的DFA构造时间.

关键词:NFA;DFA;正则表达式匹配;深度包检测

中图分类号:TP393 **文献标识码**:A doi:10.3969/j.issn.0253-2778.2014.01.001

引用格式:Peng Kunyang. A fast DFA construction algorithm by subset encoding[J]. Journal of University of Science and Technology of China, 2014,44(1):1-11.

彭坤杨. 基于状态子集编码的快速DFA构造算法[J]. 中国科学技术大学学报,2014,44(1):1-11.

A fast DFA construction algorithm by subset encoding

PENG Kunyang

(*School of Computer Science and Technology, University of Science and Technology of China, Hefei* 230027, *China*)

Abstract: Regular expression matching is the foundation of many network functions such as deep packet inspection, which is performed using either non-deterministic finite automaton (NFA) or deterministic finite automation (DFA). To meet the requirement of high-speed regular expression matching, DFA has been the prevalent choice for its deterministic nature and matching efficiency. However, all these DFA-based approaches need to construct a DFA from an NFA as an intermediate step, thus the DFA construction process can be one of bottlenecks for the system. By exploring the inherent properties of finite automaton — whether the NFA states simultaneously active and how the self-looping NFA states lead to explosion of DFA state space — a state subset encoding and searching scheme was designed, and a new DFA construction algorithm was proposed. Through experiments based on real life pattern sets from the Snort intrusion detection system, the new DFA construction algorithm was demonstrated to reduce the running time by 88.33%~93.57% compared with that of the standard subset construction algorithm.

Key words: NFA; DFA; regular expression matching; deep packet inspection

收稿日期:2013-03-26;**修回日期**:2013-04-23

基金项目:中国科学技术大学博士研究生学术新人项目资助.

作者简介:彭坤杨,男,1986年生,博士生. 研究方向:网络与系统安全,高性能计算体系结构. E-mail:pengkuny@mail.ustc.edu.cn

0 引言

正则表达式匹配(regular expression matching)技术是许多网络应用的一项核心基础技术,广泛应用于网络入侵检测与防护、恶意签名匹配、内容过滤、协议分析、基于内容的包转发等领域.该技术将待检测的模式用正则表达式表示,并深入对网络包的载荷进行匹配,以确定该网络包是否包含给定正则表达式所描述的内容.理论上,正则表达式可用与之等价的非确定性有限自动机(non-deterministic finite automation,NFA)或者确定性有限自动机(deterministic finite automation,DFA)表示[1].例如,图 1 表示的是匹配正则表达式规则 .* ab.* cd和 .* ef .* gh 的 NFA,规则.* ab.* cd 表示匹配任意多个任意字符,接着匹配 ab,再匹配任意多个任意字符,最后匹配 cd,规则 .* ef .* gh 也是相似的含义.

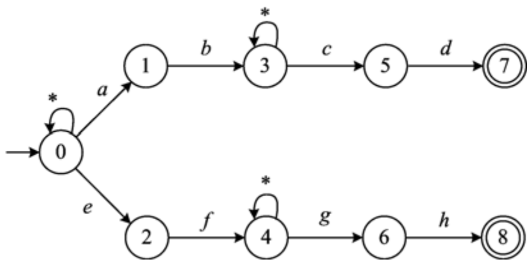

图 1 识别规则 ab.* cd 和 ef.* gh 的 NFA

Fig. 1 NFA for matching ab.* cd and ef.* gh

自动机是给定输入字符串,依据状态转移函数不断"跳转"到新的状态的一种机器,在数学意义上,自动机定义为一个五元组$\{Q, \Sigma, s_0, \delta, F\}$,这里:$Q$ 表示状态集合;Σ 表示自动机所接受的语言的字母表,即用 ASCII 码表示的 256 个字符;s_0 表示起始状态,$F\subseteq Q$ 是接受状态的集合;$\delta: Q\times\Sigma\mapsto 2^Q$ 表示状态转移函数,可用一个二维的状态转移表表示,每一行表示状态 ID,每一列表示字符,执行匹配时,将当前活跃的源状态与当前处理的字符分别作为行和列的索引,在状态转移表中查找相应的目的状态,得到的目的状态作为新的源状态参与下一个字符的状态转换.

NFA 和 DFA 这两种自动机的根本区别就在于其状态转移函数 δ:NFA 的状态对字母表 Σ 中的每个字符可以有也可以没有转移,也可以有多个转移,因此 NFA 每处理一个输入字符,就可能有多个目的状态同时被激活,而每个目的状态在处理下一个字符时又将进一步激活更多的状态.这样就导致 NFA 在执行匹配时,需要追踪不确定数目的同时活跃的状态,这造成了 NFA 的匹配效率低下和不确定.与 NFA 不同,DFA 中的状态对任意字符有且仅有一个转移,在任何时刻 DFA 中有且仅有一个状态是活跃的,因此 DFA 具有确定的、高速的匹配速度.

尽管 DFA 相比 NFA 在匹配效率上具有先天优势,但其代价是占用更多的存储空间,如图 1 的 NFA 仅 9 个状态,其等价的 DFA 却需要 16 个状态.NFA 的状态空间大小和正则表达式规则集的大小(即规则集中的字符数)呈线性关系,而在最坏情况下,DFA 的状态空间相比 NFA 的状态空间成指数膨胀.

1 相关工作

相比普通字符串匹配[2-3](pattern matching / string matching),正则表达式匹配技术更复杂,普通字符串匹配的方法[4-6]并不能有效地移植到正则表达式匹配技术中.

由于基于 NFA 和 DFA 的正则表达式匹配技术有各自的优势和缺点,研究者们基于不同的需求和平台,展开了大量有价值的研究.NFA 由于具有不确定的匹配速度,只有基于专门的并行硬件平台才能满足匹配效率的要求[7-8];网络应用对实时性的要求很高,匹配速度往往是一个不能牺牲的指标,研究者们提出了许多基于 DFA 的正则表达式匹配方法.

Yu 等[9]提出划分规则集以降低 DFA 总的空间需求.Kumar 等[10]提出了 D^2FA 的技术,即 DFA 状态之间通过共享相同的转移边来减少 DFA 的存储空间,大幅减少了每个 DFA 状态所需的状态转移边数.Becchi 等[11]提出一种改进的 D^2FA 技术,使得基于 DFA 的技术可以在压缩约 90%的空间仍然保持平摊意义上的确定性的匹配速度——处理任意一个长度为 L 的输入串,所需的状态转换次数不超过 $2L$.Chen[12]和 Meiners 等[13]相继独立地提出了基于三态内容可寻址存储器(TCAM)的 D^2FA 实现方法.DFA 的原始存储空间由状态数和每个状态的转移边数(256 条边)共同决定,以上的 DFA 压缩方法都未能突破 DFA 状态数的限制.Peng 等[14]基于 TCAM 平台,在不改变匹配效率(每处理一个字符只需一次 TCAM 查询)的前提下,首次突破 DFA

状态数这一瓶颈,使得所需的TCAM条目数小于DFA状态数.更进一步,Peng等[15]通过消除DFA空间膨胀,将DFA所需存储空间压缩到接近NFA大小.

以上所有基于DFA的方法都需要事先构造出一个DFA,然后才在这个DFA的基础上进行相应的压缩.随着网络应用的高速发展,对规则集(也即对DFA)动态更新的需求将越来越大,DFA的构造不再是一次性的脱机计算过程,因此DFA的构造过程日益成了基于DFA方法的正则表达式匹配技术的瓶颈之一.

本文要解决的问题可正式描述如下:给定一个从正则表达式规则集构建得到的NFA(本文不讨论字符串自动机[2]),如何将此NFA高效地转化为一个与之等价的DFA.子集构造算法[1]是目前从NFA构建DFA的标准算法.DFA的构造过程(也叫做确定化过程)实际上是将NFA中每一个可能同时活跃的状态集合(下文均称之为状态子集)映射成唯一的DFA状态的过程.例如图1中的NFA,初始状态子集{0}分别可以通过字符a激活目的状态子集{0,1},通过字符e激活目的状态子集{0,2},通过其他字符均激活自身{0}.如果将状态子集{0},{0,1}和{0,2}分别映射成一个用整数表示的新状态0,1和2,那么对于映射后的新状态0来说,通过任何字符都将只激活唯一的一个目的新状态(0或1或2).Leslie[16]指出,由于DFA的状态膨胀(即NFA状态子集数量的膨胀),子集构造法所需时间也随之膨胀.DFA状态空间的膨胀在实际应用中非常普遍,因此亟需改进DFA构造算法的时间性能.

2 传统算法的不足

2.1 子集构造法

传统的DFA构造算法为子集构造法[1].该算法维护一个状态子集的队列,将起始状态加入初始子集并入队,反复迭代处理队中的状态子集,通过遍历每一个输入字符,不断扩展新的状态子集,并把新扩展的状态子集加入到队列中.每一个被遍历到的状态子集实际上就代表着一个DFA状态(被映射成一个整数ID).经过反复迭代直到队列为空,即将NFA中每一个可能出现的状态子集都映射成了一个DFA状态.算法描述如下:

算法2.1 子集构造算法

输入:NFA nfa=$\{Q,\Sigma,s_0,\delta,F\}$

输出:DFA dfa=$\{Q',\Sigma,s_0',\delta',F'\}$

1. $Q'\leftarrow F'\leftarrow\emptyset$
2. $S\leftarrow\{s_0\}, p\leftarrow 0, L\leftarrow\emptyset$, $id\leftarrow 0$
3. Enqueue(L, <S, p>)
4. while $L\neq\emptyset$ do
5.   <S, p> = Dequeue(L)
6. $Q'\leftarrow Q'\cup\{p\}$
7.   if $S\cap F\neq\emptyset$
8. $F'\leftarrow F'\cup\{p\}$
9.   for each $c\in\Sigma$
10. $D\leftarrow\bigcup_{s\in S}\delta(s,c)$
11.   q←Mapping(D)
12.   if q=NIL
13.   id←id+1
14.   Mapping(D)←q←id
15.   Enqueue(L, <D, q>)
16. $\delta'(p,c)\leftarrow q$

该算法使用了一些辅助的数据结构:L是NFA状态子集S和S所映射到的DFA状态编号p的队列,id是从0开始递增分配的DFA状态编号,Mapping:$2^Q\mapsto N\cup\{NIL\}$记录的是NFA状态子集到其对应的DFA状态编号之间的映射,N表示自然数.如果Mapping(D)返回NIL,说明状态子集D尚未记录过.

2.2 性能瓶颈

本文为方便讨论,约定我们讨论的NFA大小(即状态数)为m,其对应的DFA大小为N.

在子集构造算法中,每产生一个新的状态子集D(算法2.1第10行),就需要去查询这个状态子集是否已经标记过(算法2.1第11行),如果尚未标记过,则需要将该状态子集入队(算法2.1第12~15行).下面说明整个算法的瓶颈就在于状态子集的查询.

为了高效地实现状态子集的查询,一种直观的方法是,使用数组存储映射关系,即为每一个NFA状态子集分配一个索引号,用这个索引号在数组中进行查询,即可知道每个NFA状态子集对应的DFA状态编号;但是大小为m的NFA中可能出现的状态子集数目高达2^m个,意味着这个数组需要指数级的存储空间,这是完全不可行的.另一种可能的方法就是哈希,即将高达2^m的状态子集空间哈希到一个可以接受的小空间,但由于事先无法预估实际存在的状态子集的数目(即DFA的大小),也就无法确定合适的哈希空间,从而造成哈希冲突的不确

定性.

实际上子集构造算法的标准实现[17]是用一棵前缀树(又叫单词搜索树、Trie 树)存储所有的状态子集,树中每个节点存储状态子集中的一个状态,将从根节点到某个节点所经过的状态合并起来,即为该节点对应的状态子集.将待查询的状态子集按其内部状态 ID 由小到大排序后,作为搜索字符串,即可在树中进行查询和更新.这样,共享相同前缀状态的状态子集在树中也将共享相同的存储节点.

前缀树中保存了所有的状态子集,结合前缀树的特性可知,树的存储代价为 $O(N)$.对于树中的任意节点,假设其存储的 NFA 状态 ID 为 i,则其孩子节点可能多达 $m-1-i$ 个.由于可能的孩子节点数太大,不适合将孩子节点组织成数组并按 NFA 状态 ID 进行查询,否则空间复杂度将高达 $O(m^h)$,这里 h 是树的高度,如此指数级的空间占用是不可接受的.因此,子集构造法将孩子节点组织成链表,由于遍历孩子节点的开销,使得在前缀树中查询一个状态子集的平均时间复杂度达到 $O(m)$,整个子集构造算法的迭代次数为 $O(N)$,因此总的时间复杂度高达 $O(\Sigma mN)$,这里 Σ 是常量,由于 NFA 中大部分状态都只在少数字符上有出边,因此我们可以压缩 NFA 状态的字母表,使得在子集构造时避免遍历没有转移边的字符(算法 2.1 第 9 行).字符表压缩方法与本文提出的新算法是完全互补的,本文将在 4.4 节讨论这一优化技术. N 是 DFA 大小,实际应用中,尽管 N 往往很大,但 $O(N)$ 是构造 DFA 所需要的最少迭代次数,无法减少,因此算法性能的瓶颈就在于状态子集的查询性能.对于需要频繁更新规则集从而频繁构建 DFA 的网络设备和应用来说,改进 DFA 构造的时间性能成了迫切需求.

3 基于自动机内在运行特性的新算法

为了查询一个状态子集,每次需要遍历的节点数高达 $O(m)$,如果我们对 NFA 本身的内在特性有更深入的了解,就能够消除子集查询的效率瓶颈.

3.1 状态活跃关系

状态活跃关系及状态分组的概念是由 Zu 等[7]和 Peng 等[8]提出来的,他们分别在 GPU 并行平台上和 TCAM 并行平台实现了基于 NFA 的正则表达式匹配.本文首次将基于 NFA 方法的技术引入到 DFA 的构造中.

状态分组的思想基于如下的观察:在 NFA 执行匹配的过程中,并不是任意两个状态都有可能同时活跃.例如,图 1 中的 NFA 状态 1 和状态 2,由于激活它们的输入字符不相同,因此这两个状态永远不可能同时活跃.我们可以依据 NFA 这一内在运行特性将 NFA 的所有状态划分成若干个状态分组,并且使得:在每个分组内部,任意两个状态在任何时刻都不可能同时活跃.

状态分组的算法维护一个能同时活跃的状态对的队列,并输出一个以状态对为索引的二维数组,用于指示两个状态是否能够同时活跃,该数组初始设置为任意状态对均不能同时活跃.首先将 $(0,0)$,$(1,1)$,…,$(m-1,m-1)$ 等 m 个状态对放入队列,并将这 m 个状态对设置为能同时活跃,然后不断按如下方法迭代直到队列为空:取出当前队列首元素 (i,j),分别遍历每个字符 $c\in\Sigma$,得到目的状态集合 $D\leftarrow\delta(i,c)\cup\delta(j,c)$,则显然 D 中任意两个状态 (i',j') 均能同时活跃.将所有的 (i',j') 设置为能同时活跃,若 (i',j') 尚未入队,则将其入队.

状态分组算法的时间和空间复杂度均仅为 $O(m^2)$.如果状态对 (i,j) 在算法结束后被标记为可以同时活跃,那么一定存在一个字符串,使得 NFA 从起始状态开始经过这个字符串后将同时激活状态 i 和 j;反之,如果两个状态被标记为不能同时活跃,则一定不存在任何字符串可以同时激活状态 i 和 j.由于篇幅限制,算法相关的理论证明请参考文献[7].

假定状态分组算法将状态数为 m 的 NFA,划分成了 k 个分组,即 $G_0, G_1, \cdots, G_{k-1}$,则每个分组仅需 $\lceil\log(|G_i|+1)\rceil(0\leqslant i<k)$ 个比特即可编码该分组中的每个状态(注意,每个分组需要预留一个编码表示该分组中没有任何状态活跃,我们预留编码 0).对于任一个状态子集 S,因为在任何时刻每个分组中至多只有一个状态活跃,只需将 S 在每个分组中活跃状态的二进制编码按照组号顺序依次连接起来即组成状态子集的编码(无状态活跃的分组编码为全 0).如图 2 所示,图 1 的 NFA 共 9 个状态,按状态分组算法,可分成 4 个分组:状态 0、状态 3、状态 4 各形成一个分组 G_0,G_1 和 G_2,各需要 $\lceil\log(1+1)\rceil=1$ 位比特进行状态编码;剩下的 6 个状态组成一个分组 G_3(这六个状态彼此之间都不可能同时活跃),其内的状态需要 $\lceil\log(6+1)\rceil=3$ 个比特用于状态编码,每个状态在其分组内部的二进

制编码标注在图中. 以状态子集{0}为例,该子集可编码为二进制 1|0|0|000("|"为分组间隔符,仅作演示用,下文或将略去该符号),即整数 32;再以状态子集{0,4,2}为例,该子集可以编码为二进制 1|0|1|010,即整数 42.

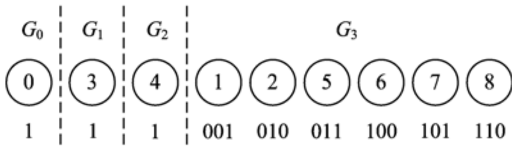

图 2　图 1 中的 NFA 的状态分组和状态编码

Fig. 2　State grouping and encoding for the NFA in Fig. 1

基于状态分组的状态子集的编码方法,使得我们仅需要编码 NFA 中可能同时活跃的状态所组成的状态子集,从而极大地降低了状态子集的空间. 基于真实规则集的实验表明,状态分组数及其编码所需比特数远远小于 NFA 状态数 m. 本文在实验部分采用的 NFA 状态数高达7 759,DFA 状态数高达876.7 万,但所有规则集编码状态分组所需的比特数都小于 64,用一个 64 位比特的整形变量即可表示.

虽然我们将状态子集可能的空间进行了极大的压缩,但考虑到编码一个状态子集所需的比特数仍有数十个比特,不适合将状态子集直接映射成整数进行寻址. 相比子集构造法用前缀树保存状态子集的集合,本文用二叉搜索树保存状态子集的二进制编码. 每查询一个状态子集,前缀树的查找时间复杂度为 $O(m)$,而二叉树的查找时间复杂度为确定的次数,即为编码一个状态子集所需的比特数.

3.2　两种不同类型的状态分组

状态间活跃关系揭示了哪些 NFA 状态才有可能同时活跃,从而参与组成 DFA 构造过程中的状态子集. 本小节将进一步探索 NFA 状态另一内在特性.

通过分析开源的网络入侵检测系统中真实的规则集可以发现,导致 DFA 状态空间膨胀的正则表达式特征主要有两类[7,9,15,18-19]:(Ⅰ)包含"点星"(dot-star)特征的规则集,正则表达式中包含形如".*"的语法,用于描述"任意字符可能出现任意多次". 包括各种形式的变种,如".+","[c_1-c_k]*","[c_1-c_k]+"等.(Ⅱ)包含"计数"特征的规则集,正则表达式中包含形如".[j]"的语法,单条这样的规则即可产生 $O(2^j)$ 的指数膨胀,而且 j 的值往往成百上千,导致这种类型的规则根本不适合用 DFA 处理,学术界现多采用文献[20-21]提出的 XFA 技术通过改造自动机语义予以解决,因此此类特征的规则不在本文讨论范围之内.

基于上述第一类导致状态空间膨胀的规则特征,我们将 NFA 状态划分为两类状态:自循环状态和非自循环状态. 这里,NFA 的自循环状态定义为超过一半的字符激活自身的状态. 如图 1 的 NFA,对于状态 0、状态 3 和状态 4,它们经过 256 个字符都会激活自身,因此这三个状态就是自循环状态,而剩下的 6 个 NFA 状态则为非自循环状态. 我们只需在状态分组算法中附加一个条件——自循环状态和非自循环状态不能分配到同一组中,就可以得到只包含自循环状态的分组(如图 2 中的分组 G_0,G_1 和 G_2)和只包含非自循环状态的分组(如图 2 中的分组 G_3),为方便后续计算,我们将自循环状态所在的分组从组号 0 开始分配连续的编号,并紧接着为非自循环状态所在的分组分配连续的编号.

自循环状态两两之间几乎都可以同时活跃,而非自循环状态两两之间极少能同时活跃. 因此 DFA 的空间膨胀主要来自于自循环状态的组合形成的状态子集(简称为自循环子集;反之,由非自循环状态形成的子集简称为非自循环子集),当然也包括自循环状态与非自循环状态的组合形成的状态子集;因此,我们可以在 DFA 构造之前预先计算和存储自循环子集的状态转移表,并在 DFA 构造过程中将状态子集分解为自循环子集和非自循环子集进行查询. 本文设计的状态子集的数据结构如图 3 所示,Mapping 是一个二叉搜索树的数组,以自循环子集的(整数)编码为索引,二叉搜索树中只需存储非自循环子集的集合. 图 3 只列出了 Mapping[5]的二叉搜索树.

下面以状态子集 S={0,4,6}为例,阐述状态子集的查询和更新步骤. 由于状态子集可直接编码为一个整数,下文在不带来歧义的前提下,可能用同一个符号表示状态子集和它的编码. 为了表示区别,下文将与自循环状态或子集有关的变量用下标 1 表示,并将与非自循环状态或子集有关的变量用下标 2 表示:

Step 1　将 S 根据状态分组和编码算法编码为一个整数(子集{0,4,2}的编码为 42(即二进制 101010)),然后将 S 分解为两个部分——自循环子集 S_1 和非自循环子集 S_2. 只需对 S 的编码做一个

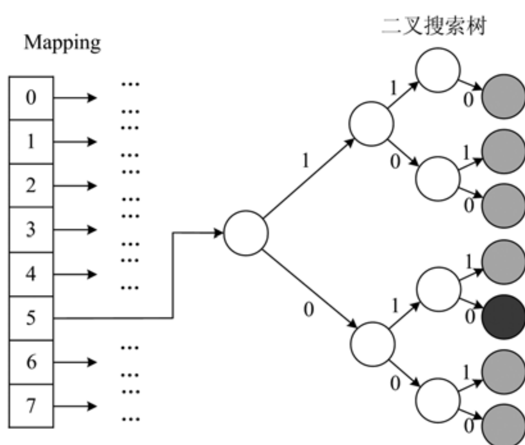

图 3　状态子集的数据结构

Fig. 3　Data structure for NFA subsets

简单的掩码运算和位移运算，即可得到 S_1 和 S_2 的编码：将 S 编码中非自循环分组部分掩码掉并移位即得到 $S_1=\{0,4\}$ 的编码为 $(101111 \& 111000)\gg 3=101$，也即整数 5；将 S 编码中的自循环分组部分掩码掉即得到 $S_2=\{2\}$ 的编码为 $(101111 \& 000010)=010$.

Step 2　进入 Mapping$[S_1]$中查询 S_2 是否已经存在，即用 S_2 的二进制编码在二叉搜索树中进行查询：如果查询失败，则在树中创建相应节点，并在最终创建叶子节点时，在叶子节点为子集 S 存储一个未分配过的 DFA 编号；如果查询成功，则在到达叶子节点时，返回子集 S 对应的 DFA 编号. 图 3 中的叶子节点用阴影标记，沿着 S_2 的编码 010 将到达图 3 的阴影加粗的叶子节点.

从上述算法不难看出，将状态子集分解为自循环子集和非自循环子集，既可将自循环子集部分直接映射成一个整数并寻址，同时又减少了二叉树中查找一个状态子集所需的访存次数，从而进一步提高了状态子集查询的性能.

3.3　DFA 构造新算法

基于 3.1 节所述状态子集的编码算法和 3.2 节的状态子集的查询算法，本小节给出 DFA 构造新算法的正式描述. 该算法使用了一些辅助的数据结构.

T_1 是算法预先计算出的自循环子集的状态转移表，任意给定一个自循环子集 S_1，$T_1[S_1][c]$即代表 S_1 在字符 c 上的目的子集(整数表示). T_2 是算法预先计算出的非自循环状态的状态转移表，为方便查询，T_2 采用三维数组表示，任意给定一个非自循环状态的组号 g 和组内编码 s，$T_2[g][s][c]$即代该状态在字符 c 上的目的子集(整数表示).

L 是 NFA 状态子集 S 以及其对应的 DFA 状态编号 p 的队列，id 是递增分配的 DFA 状态编号，Mapping 记录的是 NFA 状态子集 S 到其对应的 DFA 状态编号之间的映射.

算法 3.1　基于状态分组的 DFA 构造算法

输入：NFA nfa$=\{Q,\Sigma,s_0,\delta,F\}$

输出：DFA dfa$=\{Q',\Sigma,s_0',\delta',F'\}$

```
1.  Q′←F′←∅
2.  Group(Q)
3.  Computer(T1),Computer(T2)
4.  Allocate(Mapping)
5.  S←{s0},p←0,L←∅, id←0
6.  Enqueue(L,<S,p>)
7.  while L≠∅ do
8.    <S,p>=Dequeue(L)
9.    Q′←Q′∪{p}
10.    if S∩F≠∅
11.      F′←F′∪{p}
12.    <S1,S2>←Split(S)
13.    for each c∈Σ
14.      D←T1[S1][c]
15.      for each groupgof S2
16.        D←D|T2[g][S2[g]][c]
17.      <D1,D2>←Split(D)
18.      q←Mapping[D1](D2)
19.      if q=NIL
20.        id←id+1
21.        Mapping[D1](D2)←q←id
22.        Enqueue(L,<D,q>)
23.      δ′(p,c)←q
```

算法维护队列 L，不断取出 L 的队首元素，反复按如下步骤迭代，直到队列为空，算法结束：

Step 1(第 8 行)　取出队首元素，队首元素记录了当前要处理的 NFA 状态子集 S 和该子集对应的 DFA 编号.

Step 2(第 9～11 行)　更新 DFA 的状态空间和接收状态信息.

Step 3(第 12 行)　将子集 S 分解为自循环子集 S_1 和非自循环子集 S_2.

Step 4(第 13～23 行)　对当前子集 S，依次遍历每一个输入字符 c，计算相应的目的状态子集 D，具体如下：

Step 4.1(第 14 行)　将 D 初始化为自循环子集 S_1 的目的子集;

Step 4.2(第 15～16 行)　计算非自循环子集 S_2 中的每一个分组 g 中状态 $S_2[g]$对应的目的子集,即 $T_2[g][S_2[g]][c]$. 基于状态子集分组的原理,S_2 的目的子集中的状态必然都属于不同分组,故只需将 $T_2[g][S_2[g]][c]$与 D 做或操作,即可得到最终目的状态的并集 D;

Step 4.3(第 17～22 行)　将目的子集 D 分解为自循环子集 D_1 和非自循环子集 D_2,并在自循环子集 D_1 对应的二叉搜索树 Mapping$[D_1]$中查询是否记录了 D_2,若无记录,说明 D 尚未入队,则更新二叉搜索树 Mapping$[D_1]$(更新操作实际上在查询时已经同步完成),同时将 D_2 对应的(也即 D 对应的)DFA 状态编码记录为新增的编号($id+1$),并将新子集和新编号入队.

Step 5(第 23 行)　更新 DFA 的状态转移函数.

为节省空间,我们可以仅当某个自循环子集 S_1 第一次被使用到时才为 $T_1[S_1]$和 Mapping$[S_1]$分配空间. 为保持算法描述简洁,上文的算法伪代码中略去这一说明.

4　实验结果

本节将本文提出的 DFA 构造新算法与传统的子集构造法在运行时间和内存占用两方面进行比较. 本文实验所使用的 PC 配置为:中央处理器为 AMD A8-3870 APU(3.00 GHz),主存为 16 GB DDR3 (1 333 Hz). 本文实验采用了数百条真实的正则表达式规则,该规则集来自学术界使用最广泛的开源的入侵检测与防护系统 Snort[22](版本 2.9.2.2).

4.1　规则集

根据 Snort 系统的匹配机制,并不是所有的正则表达式规则都同时参与匹配,Snort 系统首先根据 IP 包的包头信息(协议类型、源和目的 IP 地址、源和目的端口)对规则集进行分组,只有对应的包头信息相同的规则才需要同时进行匹配. 本文实验环境提供了 16 GB 的内存空间,每个 DFA 状态 ID 需 4 个字节表示,每个状态需要存储 256 个目的状态,因此理论上可构造的 DFA 数量上限为 16 GB/4 B/256＝2^{24},考虑到算法本身开销和系统开销,本文设置状态数上限为1 000万个状态.

本文按如下方法随机生成规则集:从 Snort 规则集(已按照包头信息分组)中持续随机地取一条规则加入到当前规则集,直到再加入一条规则就会导致当前规则集生成的 DFA 状态数超过上限. 生成的 DFA 状态数小于一万的规则集不予考虑. 如此本文得到如表 1 所示的五个随机生成的规则集. 其中,Snort-89 和 Snort-232 来自具有相同包头信息(tcp, $EXTERNAL_NET, $HTTP_PORTS, $HOME_NET, any)的规则集,Snort-14、Snort-133,Snort-16 分别来自其他三个对应包头信息不同的规则集.

从规则集的复杂度和代表性来看,表 1 所示的规则集的特征统计显示出 Snort 规则具有较为丰富的复杂性:规则集名称后面的数字表示该规则集所包含的规则数量,少则十几条,多则数百条;单条规

表 1　规则集特征

Tab. 1　Characteristics of pattern sets

规则集	规则长度	平均长度	^	\x	\x+ \x*	[c1…cn] [^c1…cn]	[c1…cn]* [^c1…cn]*	OR exp
Snort-89	[33,105]	100.4	1	1\s	82 \s+ 494 \s* 1 \S+	85	85	10
Snort-232	[23,194]	100.0	0	5\s 2\d	211 \s+ 1276 \s*	218	217	19
Snort-14	[27,216]	74.4	0	1\s	2 \d+	16	15	31
Snort-133	[12.76]	31.2	14	6\d	5 \s+ 1 \s* 7 \d+	118	11	103
Snort-16	[18,105]	43.6	7	3\d 1\d	1 \s+ 14 \s* 1\d+	11	5	14

则的长度从12到216不等;每个规则集均包含一定数量的具有点星特征(\x*,[c1…cn]*)的规则,且包含了许多复杂的范围匹配(\x,[c1…cn],如\s匹配任意空白字符,\d匹配任意数字,\S匹配除空白以外的任意字符),同时锚定(^)和分支结构(OR-exp)也增加了规则集的复杂性.

从规则集生成的NFA和DFA来看,表2的第2,3,4列显示出这些规则集产生了极大的空间膨胀,这些规则集的NFA状态数从229到7 759不等,相应的DFA状态数从3万多到800多万不等,DFA状态数相比NFA状态数膨胀高达13 695.20倍(Snort-14).如此大规模的数据实验,在以往的基于DFA方法的正则表达式匹配的文献中是没有出现过的.

4.2 实现方案

本文对传统算法和新算法的实现,均是首先从正则表达式规则集生成NFA,然后从NFA开始构造DFA并记录相应的运行时间和内存占用.

本文算法的运行时间包括预处理步骤在内的执行时间,即先计算状态间活跃关系,然后基于状态活跃关系给状态分组和编码以及为自循环子集的状态转移表分配空间.状态分组和编码算法的时间复杂度和空间复杂度均为$O(m^2)$,m为NFA大小,因此实际上只占整个算法开销很小的一部分.内存占用是指算法在运行过程中占用的最大内存,包括所要构造的DFA的存储空间以及算法运行过程中不断增长的状态子集的存储空间,本文算法还包括为自循环子集预先分配的状态转移表的空间.

本文算法用64位比特长的无符号整形表示状态子集的编码,实验中实际的子集编码长度均小于64位比特,如果超出,也可以轻易扩展.

4.3 性能比较

实验结果如表2所示,第6~8列比较了两种算法的空间性能,第9~11列比较了两种算法的时间性能.

基于Snort真实规则集的结果显示,在不同大小、不同膨胀比的DFA上,相比传统算法,本文的新算法在增加较少内存空间的情况下,显著减少了DFA构造的时间,在5个规则集上,无论是生成几万个DFA状态的中等尺寸的DFA还是生成数百万的超大尺寸的DFA,新算法将DFA构造时间减少了88.33%~93.57%.实际上,两种算法占用的总内存相比DFA本身所需的内存空间均只有少量增加,这显示了这两种算法良好的空间性能.

以生成DFA状态数最多的规则集Snort-89为例,包括预处理过程在内,本文算法仅需765.75 s即可构造包含一个包含876.7万个状态的DFA,而传统算法需要多达11 904.50 s,即三个多小时才能完成,相比传统算法,本文算法增加了9.10%的内存空间,却减少了93.75%的运行时间.以生成DFA状态数最少的规则集Snort-16为例,本文算法仅需2.22 s即可构造包含一个包含3万个状态的DFA,相比传统算法较少了88.33%的运行时间.

本文进一步探索和验证两种算法性能随着规则集大小(即规则数)的增长而呈现出的趋势.上文的实验虽然包含了不同大小的规则集和DFA,但由于每个规则集包含的规则的特征都各不相同,难以有效地显示规则数量的增长对算法性能的影响.因此,本文使用一组人工合成的随机规则集来进行算法性能趋势的实验.鉴于"点星"特征的规则是产生状态膨胀的主要原因,本文随机生成多条形如".* c_1…c_{10}.* c_{11}…c_{20}"的正则表达式规则,这里"c_i"($1\leqslant i\leqslant 20$)是大小写字母和数字中的任意一个随机字符,每条规则先后匹配长度为10的两段字符串,两个字符串中间可以出现任意多个任意字符.这些特征相似的规则使得:每增加一条规则,DFA状态数就会大约翻番.本文实验用了最大数目为15条的随机规则集,此时最大的DFA包含671.7万个状态.

表2 真实规则集上的实验结果

Tab.2 Experimental results on real life rulesets

规则集	NFA状态数	DFA状态数	DFA/NFA状态膨胀	DFA占用内存空间	内存总占用			运行时间		
					传统算法	本文算法	增加	传统算法	本文算法	减少
Snort-89	3 038	8 767 233	2 885.86	8.36 GB	8.68 GB	9.47 GB	9.10%	11 904.50 s	765.75 s	93.57%
Snort-232	7 759	5 958 268	767.92	5.68 GB	5.90 GB	7.07 GB	19.86%	5 536.95 s	604.23 s	89.09%
Snort-14	393	5 382 214	13 695.20	5.13 GB	5.40 GB	6.78 GB	25.48%	5 207.88 s	454.76 s	91.27%
Snort-133	1 018	1 062 943	1 044.19	1.01 GB	1.08 GB	1.28 GB	18.67%	1 535.79 s	108.18 s	92.96%
Snort-16	229	33 300	145.41	32.52 MB	34.49 MB	38.16 MB	10.66%	19.04 s	2.22 s	88.33%

图 4 显示了算法运行时间随规则数增加的趋势. 从曲线可以看出，随着规则的增加，即 DFA 状态空间的增加，本文所需的运行时间趋势相比传统算法所需的运行时间增长要缓慢得多. 实际上，随着 DFA 构造过程中状态子集的数量的持续增加，传统算法所需的状态子集查询的访存代价越来越大，而本文算法的状态子集查询所需访存代价始终固定为非自循环子集编码的长度.

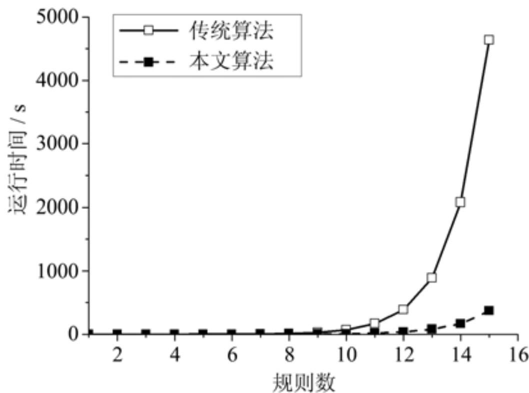

图 4　两种方法所需运行时间趋势

Fig. 4　Trend of running time required by both methods

由于两种算法所需空间主要用于存储 DFA 本身的状态转移表，因此本文在图 5 中绘制了算法空间性能随 DFA 存储空间增加的趋势图. 图 5 所绘的点表示规则数从 1 增长到 15 相应的数值，其中横坐标为 DFA 存储空间，纵坐标为算法总的内存占用. 本文对算法占用的总空间相对于 DFA 存储空间的趋势做了线性拟合，从拟合曲线可以看出，两种算法所需内存空间相比 DFA 存储空间均成线性增长，

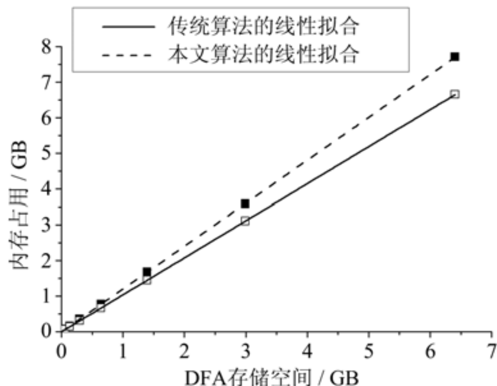

图 5　两种方法所使用的内存空间趋势

Fig. 5　Trend of memory space used by both methods

这显示了两种算法都具有良好的空间复杂度.

4.4　内存效率的优化

理论上，DFA 构造算法所需的最小空间代价为 DFA 本身所占用的存储空间，这是任何 DFA 构造算法无法逾越的空间下限，同时也是众多基于 DFA 的正则表达式匹配方法[11-15]所无法逾越的空间下限. 上一节的实验结果显示，本文算法所需要的额外空间与 DFA 本身所占的空间成线性增长，因此本文算法在未增加空间复杂度的前提下实现了对时间效率的大幅改进.

尽管本文算法具有良好的空间复杂度，但在具体实现上，若系统内存空间不足以满足算法要求，我们仍然可以引入一些辅助优化技术①以减少内存占用.

(Ⅰ) 字符表压缩技术：每个 DFA 状态在很多字符上都具有相同的目的状态，最简单地，我们可以对字符表进行游程编码，即如果连续的多个字符都共享相同的目的状态，就只需为第一个字符存储目的状态，并且记录节省存储的字符数量.

在 DFA 构造过程开始之前，我们对 NFA 的字符表同样采用游程编码的压缩技术. 在 DFA 构造过程中，对于当前 NFA 状态子集，如果它在连续的多个字符上都具有相同的目的状态子集，就只需对第一个字符进行扩展. 如算法 3.1 第 13 行所示，本文算法无需对非自循环子集 S_2 上的每一个字符都进行目的状态的扩展，因为 S_2 在某个字符上的目的状态实际上是 S_2 中每个 NFA 状态在该字符上的目的状态的并集，因此只需根据 S_2 中每个 NFA 状态的字符表上的游程，即可确定 S_2 在哪些字符上需要扩展. 一方面，上述优化方法直接产生一个压缩了字符表的 DFA，从而减少了算法的空间占用；另一方面，由于避免了遍历每一个字符，字符表压缩技术还提高了本文算法的运行效率.

需要注意的是，后续基于 DFA 的匹配算法若需要得到原始的未压缩的 DFA，只需对每个状态经过游程编码的字符表进行一次简单的还原操作即可.

(Ⅱ) 分阶段保存到文件：DFA 构造算法每生成一个新的 DFA 状态以及该状态的转移边，就可以直接存储到文件，这样就无需在内存中存储 DFA 空间. 因此这里实际上是以文件读写为代价节省了 DFA 空间所占内存.

① 这些辅助优化技术对于传统 DFA 构造算法和本文的 DFA 构造算法都是适用的，为专注于两种 DFA 构造算法本身的直接比较，本文在前面的实验中未引入这些辅助优化技术.

表 3 本文算法内存效率的优化

Tab. 3 Optimization of memory usage for our method

规则集	内存总占用			运行时间		
	优化前	优化后	减少	优化前	优化后	减少
Snort-89	9.47 GB	1.11 GB	88.32%	765.75 s	448.95 s	41.37%
Snort-232	7.07 GB	1.38 GB	80.44%	604.23 s	430.60 s	28.74%
Snort-14	6.78 GB	1.65 GB	75.70%	454.76 s	316.60 s	30.38%
Snort-133	1.28 GB	0.27 GB	78.89%	108.18 s	573.14 s	47.01%
Snort-16	38.16 MB	5.64 MB	85.20%	2.22 s	0.76 s	65.60%

本文在所有的规则集上对提出的 DFA 构造算法运用上述两种优化技术，得到优化前后的实验结果如表 3 所示. 以表 3 中最大的规则集 Snort-89 为例，该 DFA 的状态数高达 876.7 万个，优化后其构造过程占用内存仅为 1.11 GB，相比优化前减少了 85.39% 的内存占用，同时所需运行时间仅为 448.95 s，减少了 41.37% 的运行时间. 在所有的规则集上，优化后的算法相比优化前的算法减少了 75.70%～88.32% 的内存空间占用，所需内存从 5.64 MB 至 1.65 GB 不等，这些超大规模的 DFA 所需内存空间完全能满足当前多数系统的内存限制. 同时，由于字符表压缩带来的加速好处，使得优化后的算法相比优化前的算法减少了 28.74%～65.60% 的运行时间.

5 结论

本文基于 NFA 内在运行特性——NFA 状态活跃关系和 NFA 中导致 DFA 状态膨胀因素，提出了一种新的 DFA 构造算法. 算法将状态分组和编码算法应用于状态子集的存储和查询，并将状态子集分解为自循环子集和非自循环子集进行处理. 基于 Snort 真实规则集的实验结果表明，相比传统的子集构造算法，本文减少了 88.33%～93.57% 的运行时间. 基于随机规则集的实验表明，本文算法和传统算法一样具有良好的线性空间性能. 本文提出的算法打破了构造大尺寸 DFA 的瓶颈，为各种基于 DFA 的正则表达式匹配方法提供了必要的基础和更广阔的应用范围.

参考文献(References)

[1] Hopcroft J E, Motwani R, Ullman J D. Introduction to Automata Theory, Languages and Computation [M]. 3ed, Addison-Wesley, 2006.

[2] Aha A V, Corasick M J. Efficient String matching: An aid to bibliographic search[J]. Communications of the ACM, 1975, 18(6): 333-340.

[3] Boyer R S, Moore J S. A fast string searching algorithm[J]. Communications of the ACM, 1977, 20(10): 762-772.

[4] Alicherry M, Muthuprasanna M, Kumar V. High speed pattern matching for network IDS/IPS [C]// Proceedings of the 14th International Conference on Network Protocols. Santa Barbara, USA: IEEE Press, 2006: 187-196.

[5] Yu F, Katz R H, Lakshman T V. Gigabit rate packet pattern-matching using TCAM[C]// Proceedings of the 12th International Conference on Network Protocols. Berlin, Germany: IEEE Press, 2004: 174-183.

[6] Sung J S, Kang S M, Lee Y, et al. A multi-gigabit rate deep packet inspection algorithm using TCAM [C]// Proceeding of IEEE Gliobal Telicommunications Conference. St. Louis, USA: IEEE Press, 2005: 1-5.

[7] Zu Y, Yang M, Xu Z H, et al. GPU-based NFA implementation for memory efficient high speed regular expression matching [C]// Proceedings of 17th ACM SIGPLAN on Principles and Practice of Parallel Programming. New Orleans, USA: ACM Press, 2012: 129-140.

[8] Peng K Y, Dong Q F. TCAM-based NFA implementation [C]// Proceedings of the 12th ACM SIGMETRICS/Performance joint International Conference on Measurement and Modeling of Computer Systems. London, UK: ACM Press, 2012: 379-380.

[9] Yu F, Chen Z F, Diao Y L, et al. Fast and memory efficient regular expression matching for deep packet inspection [C]// Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems. San Jose, USA: ACM Press, 2006: 93-102.

[10] Kumar S, Dharmapurikar D, Yu F, et al. Algorithms to accelerate multiple regular expressions matching for deep packet inspection [C]// Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer

Communications. Pisa, Italy: ACM Press, 2006: 339-350.

[11] Becchi M, Crowley P. An improved algorithm to accelerate regular expression evaluation [C]// Proceedings of the 3rd ACM/IEEE Symposium on Architecture for Networking and Communications Systems. Orlando, USA: ACM Press, 2007: 145-154.

[12] Chen M. TCAM-based high speed regular expression matching[D]. University of Science and Technology of China, Hefei, China, 2010.

[13] Meiners C R, Patel J, Norige E, et al. Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems [C]// Proceedings of the 19th USENIX conference on Security. Washington, USA: USENIX Association, 2010: 8.

[14] Peng K Y, Dong Q F, Chen M. TCAM-based DFA deflation: A novel approach to fast and scalable regular expression matching [C]// Proceedings of IEEE International Workshop on Quality of Service. San Jose, USA: IEEE Press, 2011: 1-3.

[15] Peng K Y, Tang S, Chen M, etal. Chain-based DFA deflation for fast and scalable regular expression matching using TCAM[C]// Proceedings of the 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems. Brooklyn, USA: ACM Press, 2011: 24-35.

[16] Leslie T. Efficient approaches to subset construction [D]. University of Waterloo, Ontario, Canada, 1995.

[17] Becchi M. Regular expression processor [EB/OL]. http://regex.wustl.edu/.

[18] Becchi M and Crowley P. Efficient regular expression evaluation: Theory to practice[C]// Proceedings of the ACM/IEEE Symposium on Architecture for Networking and Communications Systems. San Jose, USA: ACM Press, 2008: 50-59.

[19] Becchi M, Franklin M A, Crowley P. A workload for evaluating deep packet inspection architectures [C]// Proceedings of International Symposium on Workload Characterization. Seattle, USA: IEEE Press, 2008: 79-89.

[20] Smith R, Estan C, Jha S. XFA: Faster signature matching with extended automata [C]// Proceedings of the IEEE Symposium on Security and Privacy. Oakland, USA: IEEE Press, 2008:187-201.

[21] Smith R, Estan C, Jha S, et al. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata [C]// Proceedings of the ACM SIGCOMM Conference on Data Communications. Seattle, USA: ACM Press, 2008: 207-218.

[22] Snort: A free and open source network intrusion detection system (NIDS) and network intrusion prevention system (NIPS) [EB/OL]. http://www.snort.org/.