Chapter 2
Related pattern-matching algorithms

Generally, pattern-matching algorithms can be classified into two different approaches- Wu-Manber algorithm and automata data structure [6-7]. Wu-Manber algorithm [8] has better performance in average case, but it is very slow in worst case. Choosing automata approach to solve our problem is the suitable way to avoid the bottleneck with low performance in worst case.

2.1 Aho-Corasick pattern-matching algorithm

Aho-Corasick pattern-matching algorithm is a famous automata approach for multi-pattern matching [9-12]. In AC algorithm, it could check other patterns simply when a mismatching event occurs during matching. AC algorithm matching automata for a given finite set *P* of patterns are the finite automata *G*, accepting the set of all words containing a word of *P* as a suffix. *G* consists of the following components.

- Finite set Q of states
- Finite alphabet *A*
- Transition function g: $Q \times A \rightarrow Q + \{\text{fail}\}$
- Failure function h: $Q \rightarrow Q + \{\text{fail}\}$
- Initial state q_0 in *Q*
- A set *F* of final states

The transition function

Let $g: Q \times A \rightarrow Q + \{\text{fail}\}$ denote the transition function of deterministic finite automaton (*A*, *Q*, *init*, *g*, *T*), where *A* is an input alphabet, *Q* is a finite set of states, *init* is the initial state, and *T* is the set of terminal states. The value *g(q,a)* is the state

reached from state *q* by the transition labeled by the input symbol *a*.

The *failure function*

Let *Q* be the set of states of Aho-Corasick automaton and let *h: Q -> Q* + {fail} denote the failure function. Let *q, q'* be states of *Q and h (q) = q' iff among the states of Q, q' delivers the longest true suffix of path (q)*

For example, an AC automata for patterns = {ab, ba, bab, bb} is illustrated in Figure 2-1.

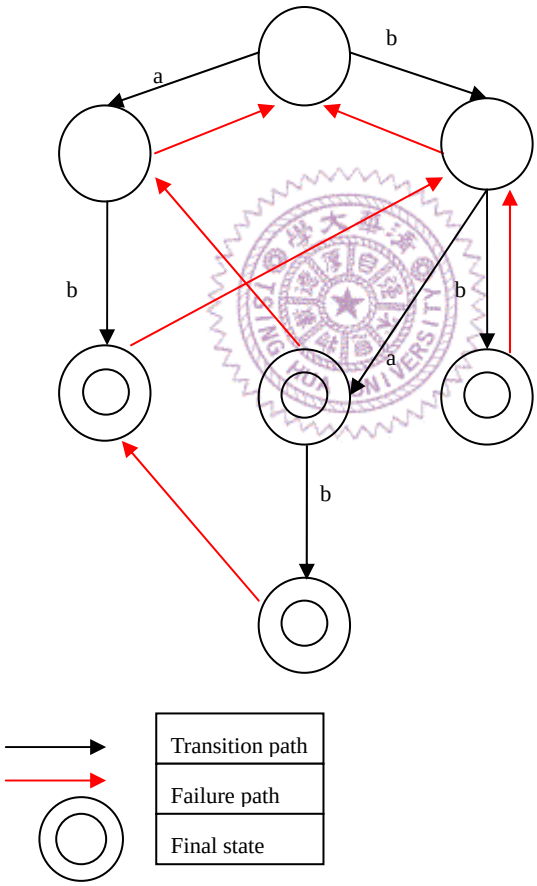


Figure 2-1 An example of AC automata.

2.2 EGREP matching algorithm

AC algorithm could only handle “exactly matching” operations but there are more and more complex operations and relationships between patterns in Snort rules. We need something to process those rules with complex operations written in Snort rule format or Regular Expression. EGREP is an extension version of GREP which is custom for pattern-matching in Unix System [13]. It could handle extension Regular Expression patterns and report if any of those patterns is matched in input string.

The data structure in EGREP is linear encoding of NFA and every node in automata consists of one byte-length operation-code (op-code), two byte-length pointers, and an optional operand if necessary. There are two main functions in EGREP – one is *regcomp*, which compiles input rules and patterns then constructs automata, and the other is *regexec*, which executes matching process in automata.

- **All functions in *regcomp***:

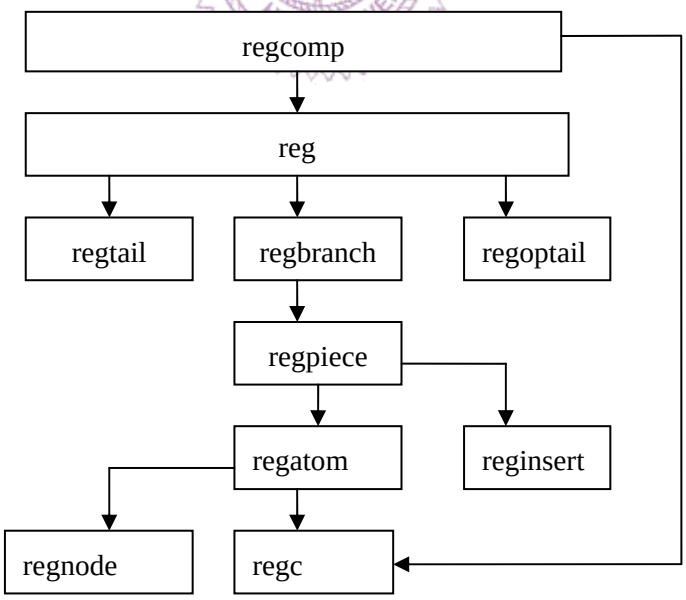


Figure 2-2 Function diagrams of EGREP compiling

- **All functions in *regexec:***

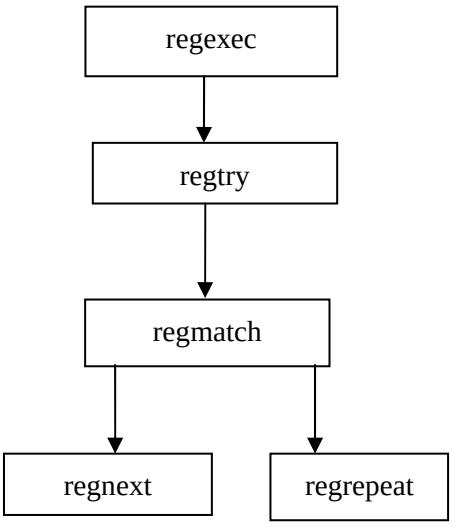


Figure 2-3 Function diagrams of EGREP matching

PCRE [14] is another Regular Expression version that Perl language supports. Although there are some formats different from standard Regular Expression, it also uses GREP kernel to process Regular Expression matching, just like EGREP. In Snort rules, those content patterns started at “pcre:” mean that this pattern is written in PCRE.

2.3 Comparison between AC and EGREP

Some comparisons between AC and EGREP are illustrated in Table 1.

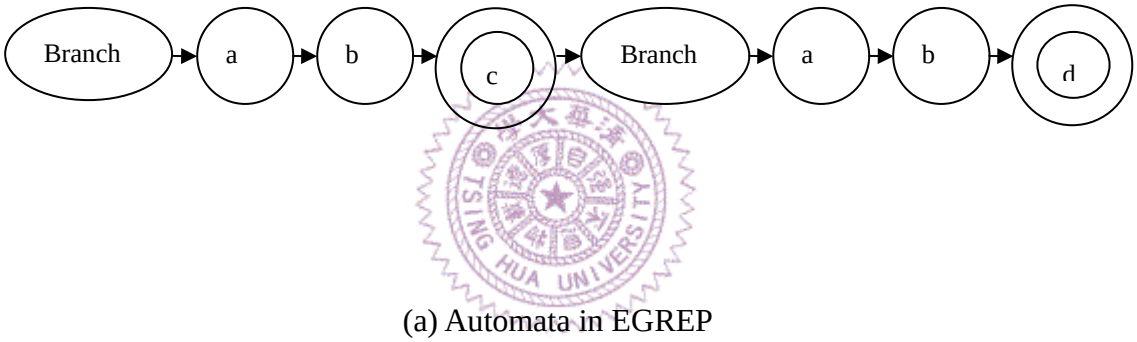
Table 1 Comparisons between EGREP algorithm and AC algorithm

	EGREP	***AC***
Algorithm	DFA	AC

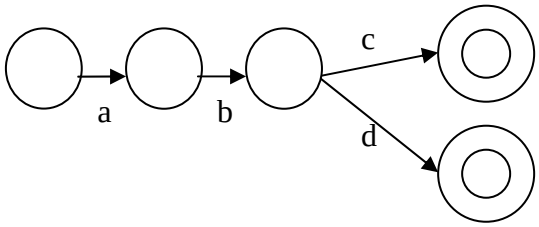
Regular Expression	Support	Not support
Multi-pattern	Not good	Good and easy
Performance	Better	Worse
low relativity data	Smaller size	Bigger size
high relativity data	Bigger size	Smaller size

And there are some examples below to show the differences in the constructed data structure between AC and EGREP when input the same rule.

If patterns = {abc, abd}, the automata built for EGREP and AC automata are depicted as Figure 2-4.



(a) Automata in EGREP

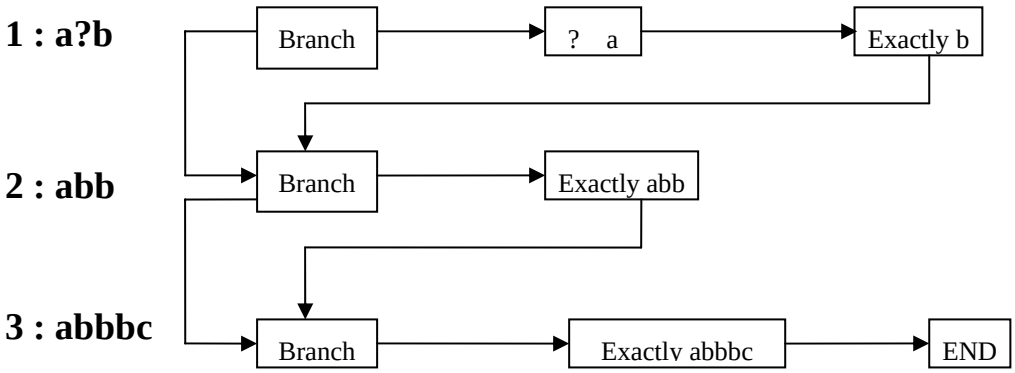


(b) Automata in AC

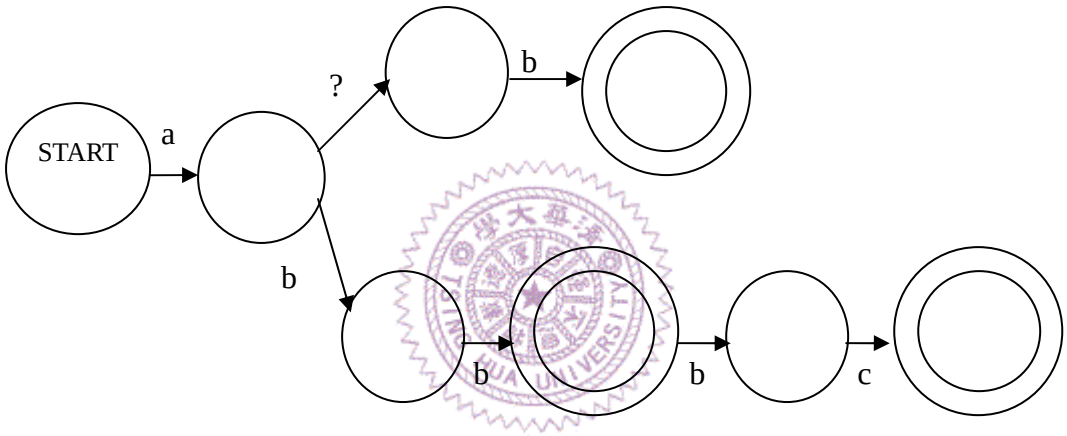
Figure 2-4 Examples of EGREP and AC automata

If rules = { a?b, abb, abbbc} and input = { aaaa, abbbb, abab, a, b, babbbc}, EGREP will find inputs 2, 3, 5, 6 matched and AC will only find inputs 2, 6 matched.

The data structures in EGREP and AC are shown in Figure 2-5.



(a) Data structure in EGREP



(b) Data structure in AC

Figure 2-5 Examples of data structure in EGREP and AC automata

EGREP has ability to handle Regular Expression and its data structure is like linked-list chain. The data structure in AC algorithm is tree-like, for multi-pattern matching, and it only handles “Exactly Matching”. We need an algorithm that supports Regular Expression and could do multi-pattern matching with high performance and low storage space requirements.