Chapter 4
Automata for Regular Expression

In newly Snort rules, more and more rules are written in Regular Expression. Single-pattern algorithms such as GREP and PCRE are employed in Snort. This is a direct solution but may not good enough. It is interesting that if an automata can do multi-pattern matching not only for common strings but also for Regular Expressions, then we have the chance to improve the performance of handling Regular Expression.

4.1 The architecture in Regular Expression automata

In our Regular Expression automata, the architecture is partitioned into three parts, a compiler for compiling all the entered rules, a data-chain in EGREP for handling those Regular Expression patterns output by the compiler, and the main automata in AC algorithm for doing multi-pattern matching, as shown in Figure 4-1.

.

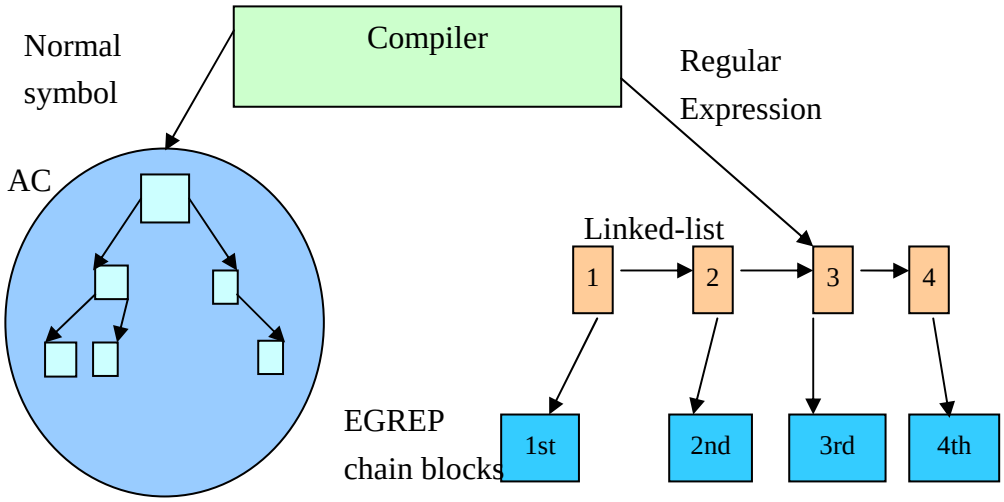


Figure 4-1 Overview of Regular Expression Automata

With this new automata, we then could process some complex rules in Regular Expression by EGREP and doing multi-pattern matching in AC automata with better performance. This new automata has both the efficiency of AC algorithm and the ability of EGREP to solve Regular Expression.

For demonstration, the data structure in Regular Expression automata for two rules = {ab[i-k]+d, cd[i-k]+a} is shown in Figure 4-2.

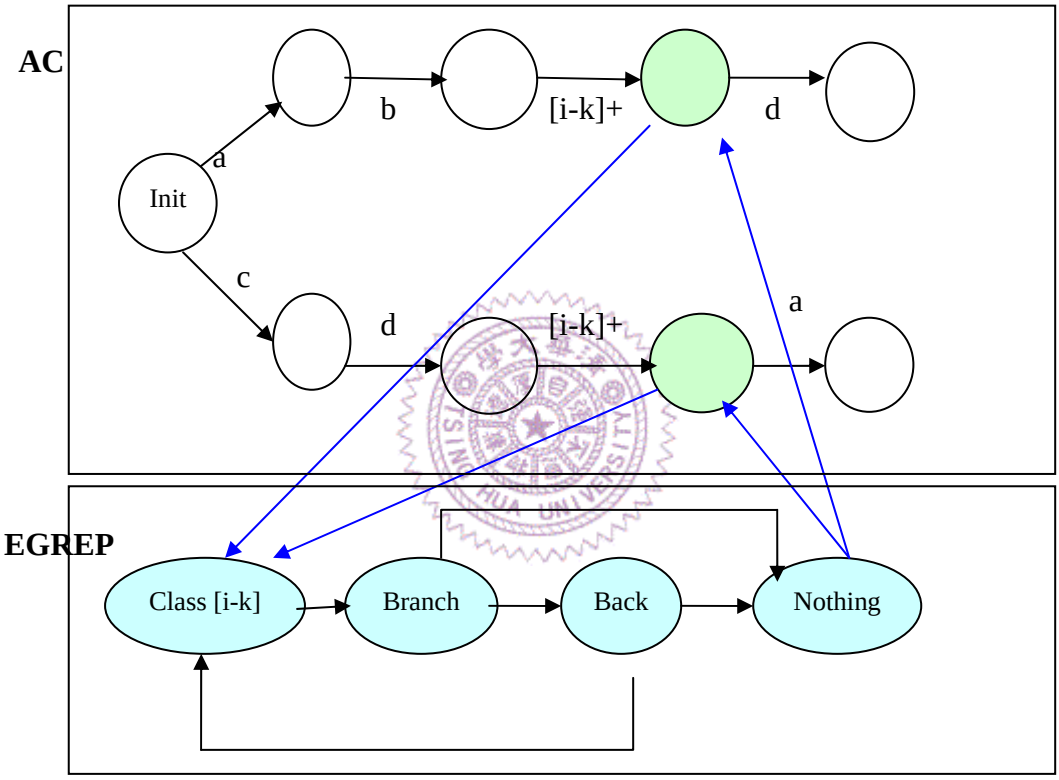


Figure 4-2 An example of data structure in Regular Expression Automata

In matching phase, the input is processed in AC automata step by step. When enters a Regular Expression state (green one in the AC automata of Figure 4-2), which means this operation is a complex one in Regular Expression, link to EGREP data structure for processing this special operation. Then go back to AC automata after the EGREP is walked through and finish the rest steps accordingly.

4.2 Automata in automata

The idea mentioned above is similar as “automata in automata”. Every node in automata may be another small automata structure and those automata have their own partial jobs to process. For example, the concept of automata in automata is shown in Figure 4-3. With scalability, we could change those detailed automata with the same big automata architecture when we want to add some new operations or functions.

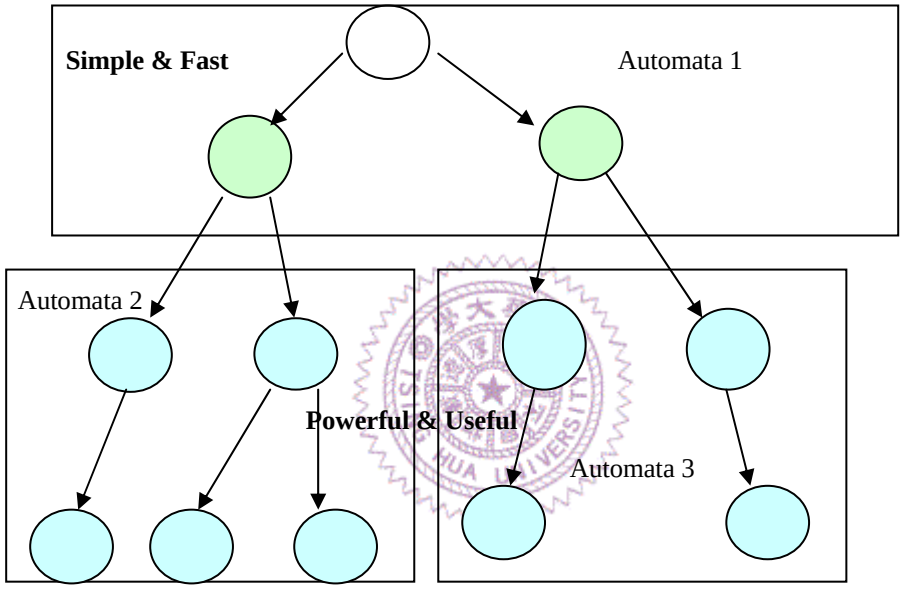


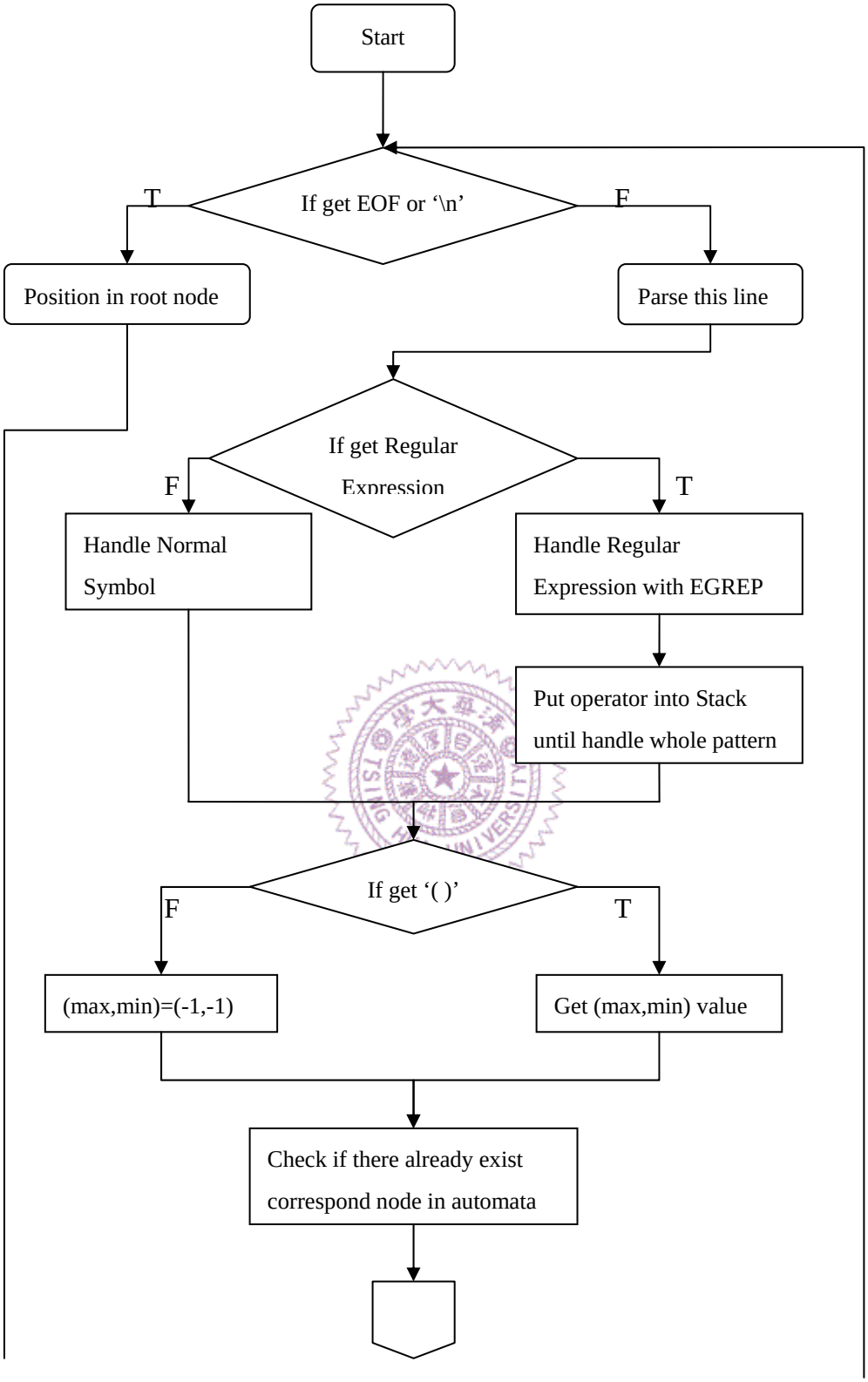
Figure 4-3 Concept and example of automata in automata.

4.3 Implement Regular Expression automata

In the implementation, the whole system is divided into two steps: Compiling and Matching. In the first step, the input rules are compiled into unique format, and the corresponding automata from these rules is constructed. After all the rules are compiled, build all failure paths in the automata to support multi-pattern matching. For the second step (matching step), take every symbol of data as the input of the transition function in automata. For unmatched input, the failure path is used to check

other patterns. After matching process is completed, obtains the results of those matched rule IDs.

The compiling and matching flowcharts of our Regular Expression automata are depicted in Figure 4-4 and Figure 4-5, respectively.


Start
If get EOF or '\n'
T
F
Position in root node
Parse this line
If get Regular Expression
F
T
Handle Normal Symbol
Handle Regular Expression with EGREP
Put operator into Stack until handle whole pattern
If get '( )'
F
T
(max,min)=(-1,-1)
Get (max,min) value
Check if there already exist correspond node in automata

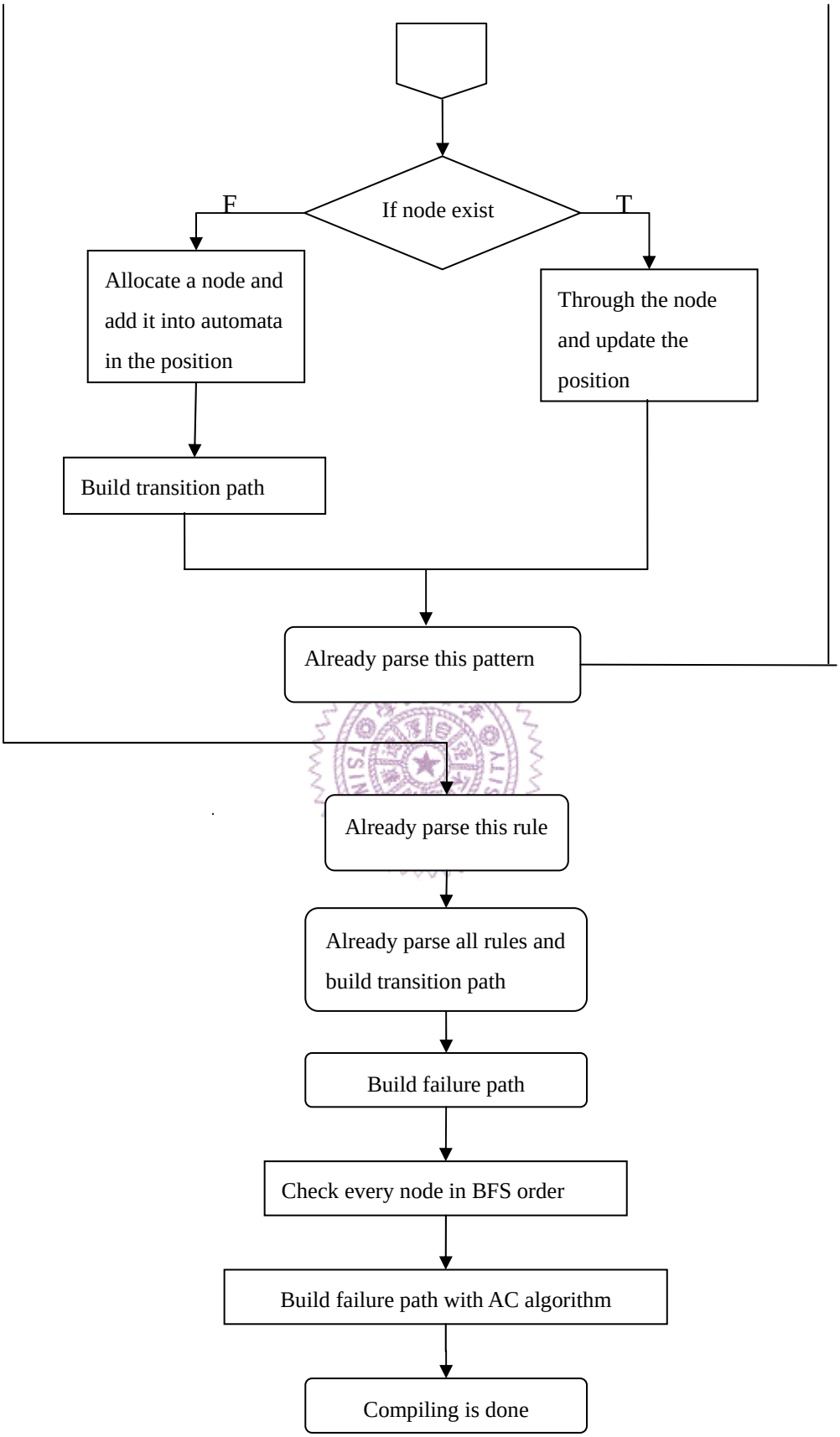
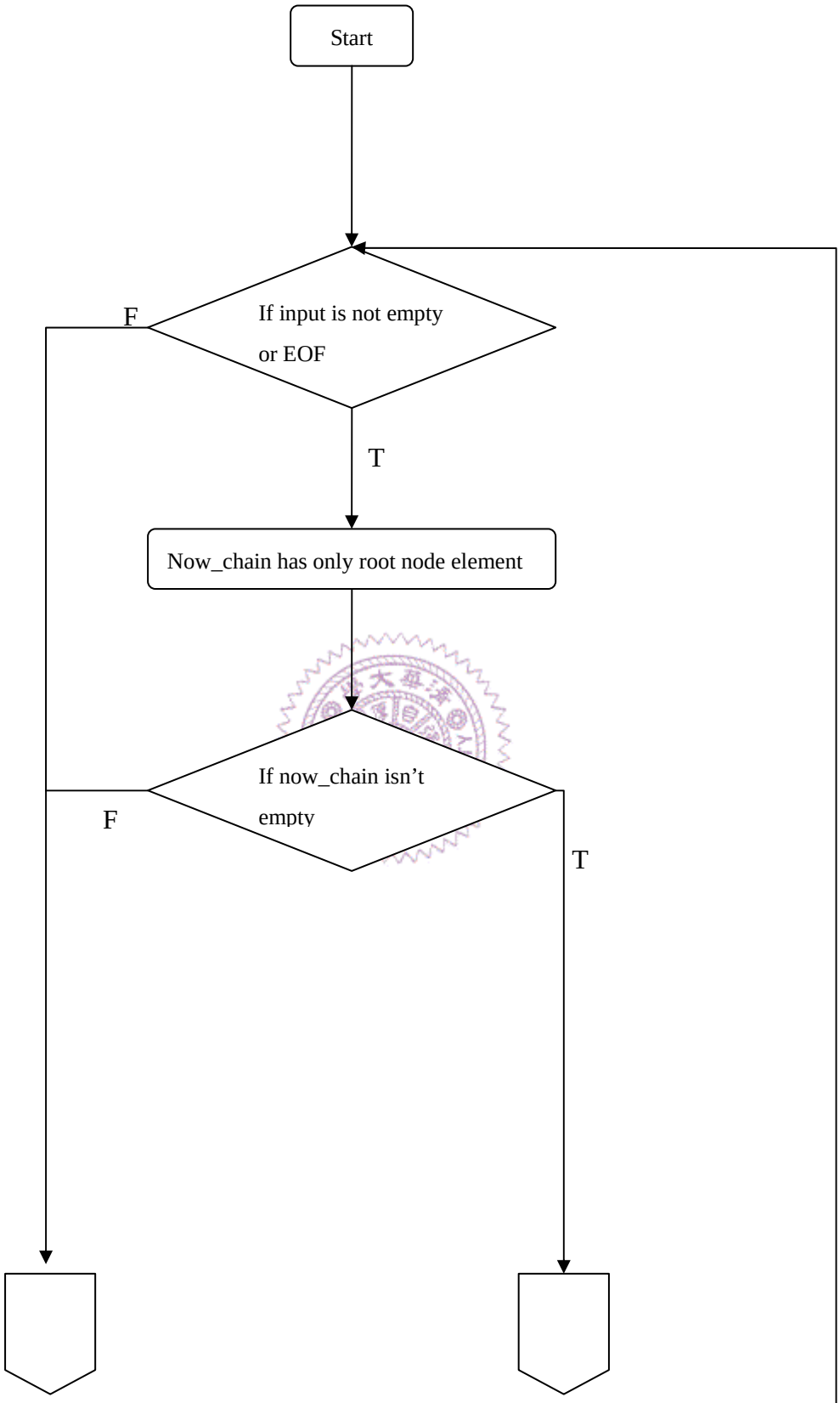


Figure 4-4 Compiling Flowchart of Regular Expression automata


Start
If input is not empty
or EOF
F
T
Now_chain has only root node element
If now_chain isn’t
empty
F
T

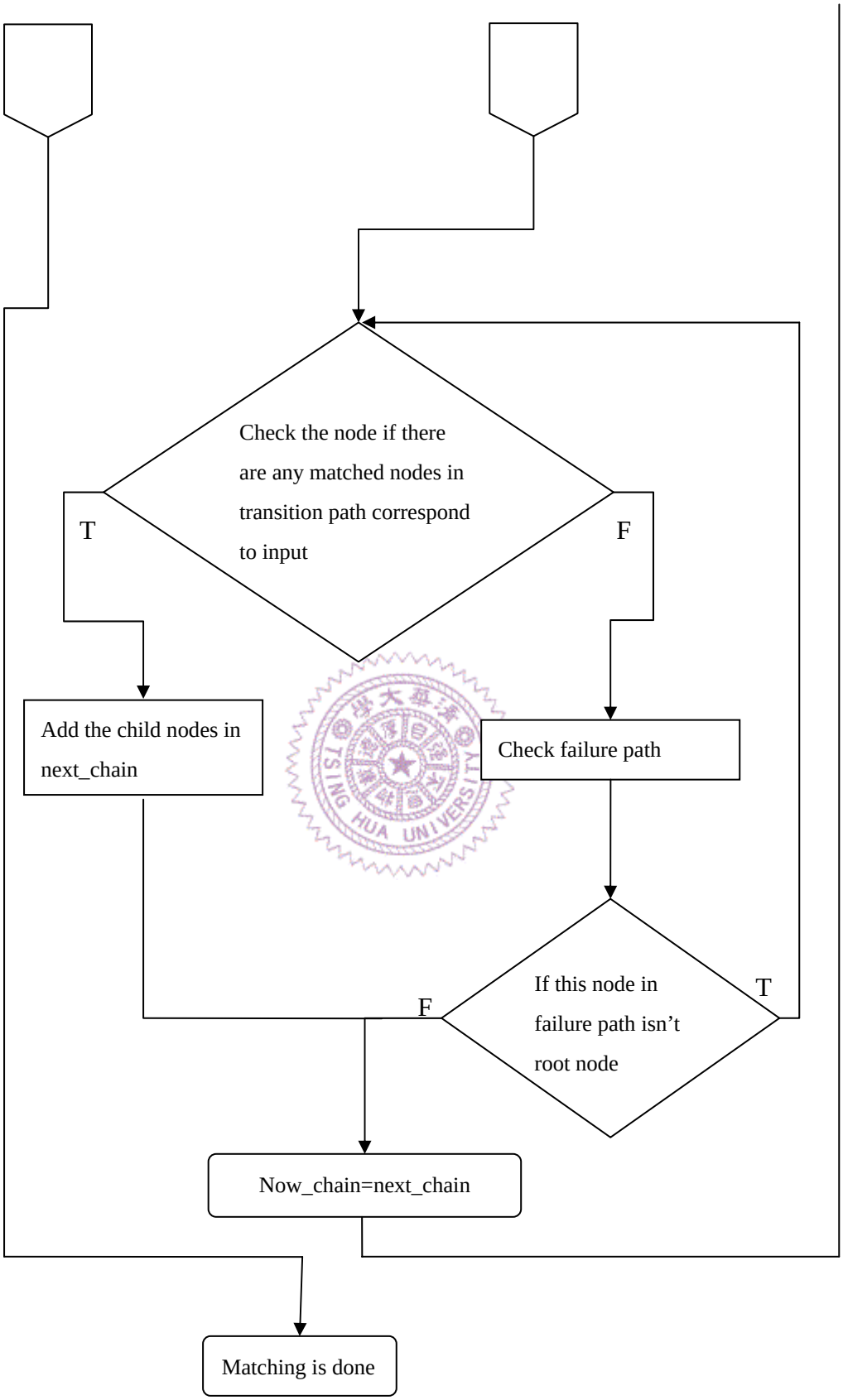


Figure 4-5 Matching Flowchart of Regular Expression automata.

4.4 Compiler and Optimization

In order to save data space and make our automata as tiny as possible, a good compiler is required to optimize the data structure and reuse them while parsing. Thus, when a Regular Expression pattern or normal pattern is parsing, we need to check if there exists the same structure to reuse. Then consider the way to divide a long complex Regular Expression pattern to make a corresponding smaller and efficient structure. Some optimizing processes are also applied after the automata is constructed. There are several articles addressed the ways for optimizing pattern-matching algorithm [16-20] or automata [21-24] to save storage space and raise the performance.

4.5 Experiments and comparisons between Regular Expression automata and EGREP

To evaluate the performance of the proposed Regular Expression automata, some experimental tests are designed. The data size conducts by the automata with some random short rules is evaluated on a PC with 400 MHz CPU.

For some **low-relativity patterns** P = {abcdef, ghghklnl, opkahqw, nkloif, Hlna}, the data size of EGREP is 70 bytes, and that of Regular Expression automata is 1140 bytes. For **high-relativity patterns** P = {aaaaaaaaaaaaaaaaaaaaaaabc, Aaaaaaaaaaaaaaaaaaaaaaaaade, aaaaaaaaaaaaaaaaaaaaaaafg, aaaaaaaaaaaaaajk, aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaalm}, the data size of EGREP is 176 bytes and that of Regular Expression automata is 1644 bytes. But for **more high-relative** patterns: 18 rules which all begin with ‘aaaaaaaaaaaaaa’, the data size of EGREP is 688 bytes and that of Regular Expression automata is 1752 bytes.

Form this trend, we can see that if there are more high-relativity patterns, one day the size of EGREP may be bigger than that of Regular Expression automata. If input patterns are high-relativity or there are lots of rules, using our Regular Expression automata is the better choice. Otherwise, using EGREP could support RE and save more space.

Then we test Regular Expression automata with those rules only include content parts which our automata support. If we input **79** k bytes-length string as rules mixed with normal words and Regular Expressions, the data size of EGREP is **129** k bytes and that of Regular Expression automata is **230.6** k bytes.

Then different lengths of strings are input to test the matching time of the proposed Regular Expression automata and EGREP. The result is shown in Table 5 and Figure 4-6. We can see that our Regular Expression automata is faster than EGREP. This is because that the Regular Expression automata has the ability to do multi-pattern matching in the same time, but EGREP just does matching in sequence. Therefore, Regular Expression automata is the more suitable solution to process Regular Expression patterns, especially in our hardware design as a Regular Expression engine.

Table 5 Comparison of matching time (Regular Expression vs. EGREP).

Input length (bytes)	Matching Time (time ticks)	
	Regular Expression Automata	EGREP
100	80	120
200	270	341
300	341	481
400	501	671

500	600	810
600	701	991
700	821	1122
800	910	1302
900	1052	1492
1000	1182	1612
1100	1242	1803
1200	1362	1953
1300	1482	2042
1400	1623	2233
1500	1702	2364

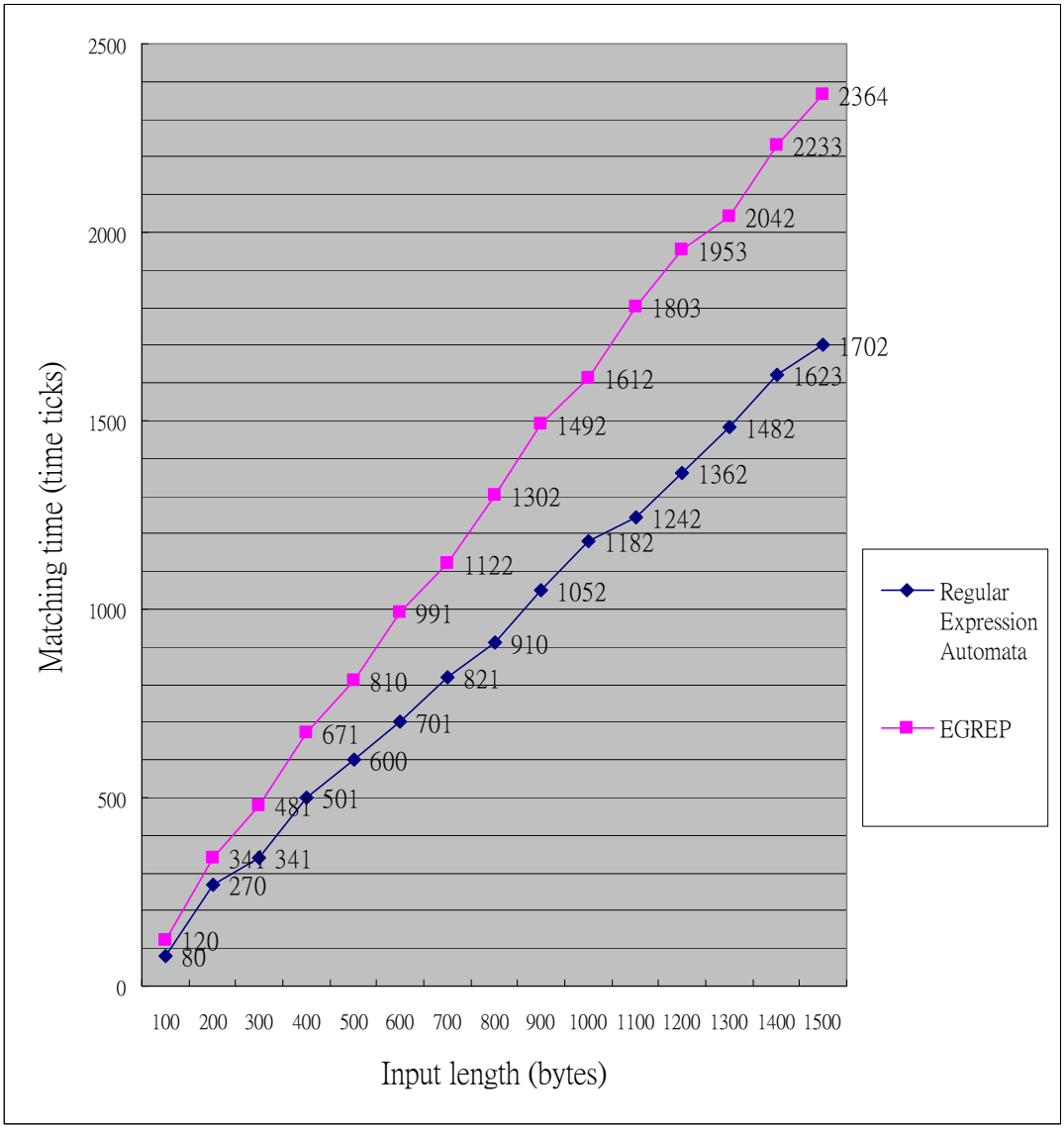


Figure 4-6 Comparison of matching time for different data sizes