A Framework for the Dynamic Implementation of Finite Automata for Performance Enhancement

Ernest Ketcha Ngassam[1], Bruce W. Watson[2], and Derrick G. Kourie[2]

[1] School of Computing, University of South Africa,
Pretoria 0003, South Africa

[2] Department of Computer Science, University of Pretoria,
Pretoria 0002, South Africa

e-mail: ngassek@unisa.ac.za, {bwatson, dkourie}@cs.up.ac.za

Abstract. The aim of this work is to provide a model for the dynamic implementation of finite automata for enhanced performance. Investigations have shown that hardcoded finite automata outperforms the traditional table-driven implementation up to some threshold. Moreover, the kind of string being recognized plays a major role in the overall processing speed of the string recognizer. Various experiments are depicted to show when the advantages of using hardcoding as basis for implementing finite automata (instead of using the classical table-driven approach) become manifest. The model, a dynamic algorithm that combines both hardcoding and table-driven is introduced.

Keywords: Finite Automata, Hardcoding, Performance

1 Introduction

To the best of our knowledge, hardcoding of finite automata (FAs) in the context of right linear languages was first suggested by Knuth et al in [Kmp77]. An intensive investigation of the behavior of hardcoded FAs in comparison with the traditional table-driven approach was suggested by Ketcha in [Ket03]. A conclusion of Ketcha's work was that hardcoding outperforms table-driven up to some threshold. This threshold is clearly dependent on such factors as processor configuration, alphabet size and number of states of the FA under consideration. In fact, for the hardware configuration used in Ketcha's experiments, it has been found that a threshold at about 360 states is relatively robust for alphabet sizes in the range of 40 to 80. For smaller alphabet sizes it is less than 1000 states. In principle, therefore, experimentation can be used to derive a set of rules to determine an estimate of the breakeven point between these two implementation strategies.

In this paper, we propose the notion of dynamic implementation of FAs to enhance performance. Table-driven (TD) implementation of large FAs often implies some latency due to insufficient memory. The problem is then to design and implement an algorithm that takes into consideration the size of the automaton upon which the recognizer is based and also the kind of string being tested for acceptance. The end result

of such an algorithm is to provide a flexible and powerful tool that takes advantage of the strengths of both table-driven and hardcoded techniques for the construction of optimal recognizers. This paper introduces the idea of dynamic Implementation of FAs for Performance (DIFAP). Unlike the traditional statically based approach that always yields to some latencies, DIFAP aims to provide a highly flexible framework enabling implementers to gain considerable time. Of course, factors such as hardware and operating system capabilities may constitute a bottleneck for efficient implementation of FAs. However, in this preliminary work, we only consider the kind of string as well as the automaton size in the design of DIFAP.

Using the gnu C++ compiler and Netwide Assembler (NASM) to encode table-driven and hardcoded algorithms respectively, we perform various experiments in order to capture the advantage of using either approach under specific considerations. All the experiments were conducted under the Linux operating system on an Intel Pentium 4 with 512 MB of RAM.

The structure of the remaining part of this paper is as follows: in Section 2, we report on the performance of randomly generated FAs over randomly generated strings. This illustrates that the threshold of efficiency of hardcoding over table-driven implementation is relatively independent of the alphabet size. Section 3 investigates some string patterns where hardcoding implementation always outperforms the table-driven algorithm. In Section 4, we introduce the design of DIFAP as the basic algorithm for efficient implementation of FAs under any circumstances. Section 5 provides a conclusion of the work and points to future directions to be taken in further investigating and implementing DIFAP.

2 Experiments based on random strings

Experiments have been conducted, based on the random generation of 100 different automata of size varying between 10 and 1000 states. For each such group of automata generated, the alphabet size under consideration was respectively, 10, 15, 20, 25, 30, 35, 40, 45, and 50 symbols. For each automaton of size n that was generated, a random accepting string of size $n-1$ was also generated. (We only relied on accepting strings instead of rejecting strings since the latter are most likely to require less time to be processed by the recognizer.) This randomly generated string was processed 50 times. The minimum, maximum and average time in clock cycles (ccs) to process the string over the 50 processing cycles were collected. In order to prevent side effects due to operating system and CPU overheads, the data relating to minimum processing times was considered to be the most accurate metric upon which further experimentation should be based. This minimum time metric was then divided by its number of symbols processed in order to estimate the time required to accept a single symbol. Such an approach was first suggested in [Kwk03a]. The collected data was then plotted to visualize the effect of the recognizer on randomly generated strings.

Figure 1 depicts the graphs for 10 and 50 symbols alphabet related to the table-driven experiments. The figure clearly shows that as the alphabet size grows, the processing time per symbol grows as well.

The hardcoded experimental results are depicted in figure 2. The generated hardcode (HC) relied on the approach suggested in [Kwk03a] using a jump table to hardcode each entry of the transition matrix. The hardcode is structured in blocks of

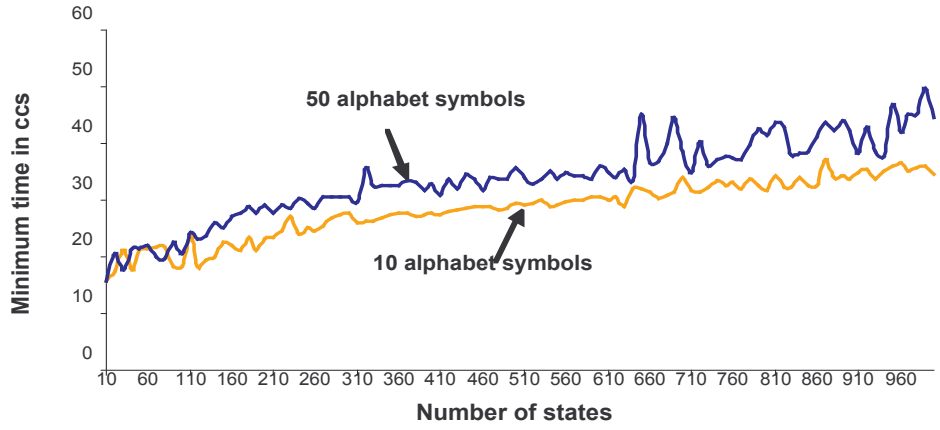

Figure 1: Performance based on table-driven experiments using random strings

assembly code, each block relating to a given state. These blocks are arranged contiguously in memory and in previous hardcode experiments, the order of the blocks corresponded to the order of rows in the transition table as shown in Source code 1. In those experiments, the strings tested were such that the flow of control through the assembly code caused jumps between contiguous blocks. Here, the same strings were used. However, to simulate the effect of a "random" string of a given length, the blocks were arranged in memory in a random order, so that jumps now take place to random places in memory[1]. The plotted graphs clearly show that for an automaton based on a 10 symbols alphabet of size 1000 states, the average time required to accept a symbol is about 70 ccs. This time is somewhat worse for an automaton of the same size based on 50 symbols alphabet (approximately 350 ccs). A plausible explanation for this is that the time is related to cache misses, due to the growth in the code size as the alphabet size grows, since the number of instructions required to encode the jump table becomes considerably larger. However, in the graphs, we can observe that in the region of 10 to 360 states, the average processing speed is always less than 50 ccs, irrespective of the number of alphabet symbols under consideration. It appears that in this range, the size of the code can still fit into the hardware's cache memory, and therefore results in efficient processing.

Source-code 1 *Extract hardcoded implementation of a recognizer using NASM*

```
asm_main:
...
mov edx, string
STATE_0:
   ;test if more symbol to process
```

[1]The notion of a random or average string is somewhat ill-defined. Nevertheless, the time taken to recognize such strings characterizes the hardcode behaviour for input that lies somewhere between best and worst case.

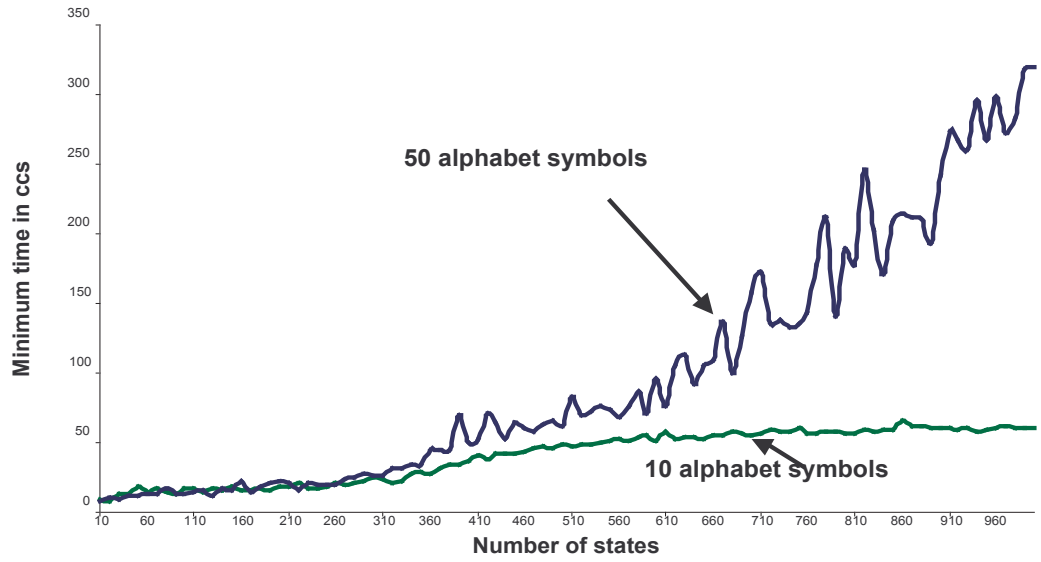

Figure 2: Performance based on hardcoded experiments using random strings

```
   jne  ACCEPT     ; no more symbol to process
   ;get the current symbol from the string
   mov      ebx, edx
   inc edx  ; points to the next symbol for further processing
   ; get the state where the symbol transits to
   shl     ebx, 2
   mov     esi, edx
   add     esi, ebx
   mov     ebx, [esi]
   shl     ebx,2
   mov     esi,ST_0   ; Label to access appropriate transition entry
   add     esi,ebx    ; related to the symbol being tested
   jmp     [esi]      ; jump to the appropriate entry of the transition table
   ST_0_TR0:          ; transition for the first symbol of the alphabet
    jmp STATE_1
   ST_0_TR1:          ; transition for the second symbol of the alphabet
    jmp STATE_1
   ST_0_TR_2:
    jmp reject        ; no transition (rejecting symbol)
   ...
STATE_1:
   ...
REJECT:
      ;do action reject
ACCEPT:
      ;do action accept
```

In order to more concisely observe the threshold of efficiency of one approach over another, we subtracted each table-driven data item from the corresponding data item of its hardcoded counterpart. This allowed us to capture the extent to which hardcode outperforms table-driven, or otherwise. Figure 3 shows the resulting plotted graphs. It clearly shows that for each automaton of a given alphabet size, the threshold of efficiency of the hardcoded version over the table-driven version is in the region of about 360 states. Such a result suggests that for any automaton based on any number of alphabet symbols in the range tested, the hardcode implementation is, on average, faster than the table-driven implementation if the number of states that make up the

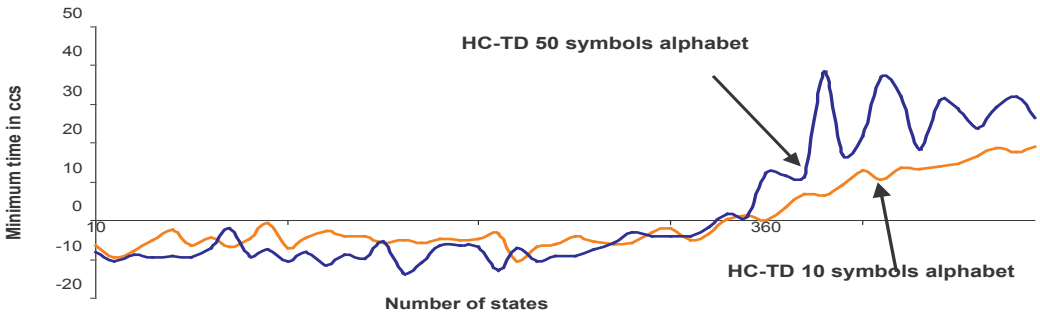

Figure 3: Performance comparison between hardcode and table-driven

automaton is less than 360. This is apparently due to the fact that, in this range of states, the hardcode is (mostly) in cache at run-time whereas table-driven processing is subject to random memory accesses. Above about 360 states, hardcode instructions from the main memory have to be accessed more frequently, and consequently there is a degradation in the average per state processing speed of the hardcode recognizer. The rate at which cache misses occur appears to be quite heavily dependent on the alphabet size. There is also a gradual degradation in the average per state processing speed of the table-driven recognizer. The behaviour in the table-driven is slightly more stable and seemingly less critically dependent on alphabet size. The table-driven code fits comfortably into cache, and so the issue of cache misses at the code level does not arise. The degradation in performance is therefore primarily due to the effect of data cache misses, the rate of which increases as the table size grows.

These observations allow us to project the worst-case scenarios for the two implementation strategies. These will occur when each and every state transition results in a cache miss. In the case of the table-driven recognizer, this will be a data cache miss, and in the case of the hardcode recognizer – a code cache miss. It is not obvious how experiments could be constructed to simulate this precise behaviour. However, it is not necessary to do so, for the above experiments already indicate the relative merits of the two approaches when a high rate of cache misses occur: the table-driven recognizer is then faster.

An interesting consequence of such an observation is the fact that, using the present threshold of efficiency, we can explore within that region various string patterns that may be subject to high hardcoded efficiency without having to restrict ourselves to the alphabet size and the size of the automaton being processed. The next section discusses such strings.

3 Experiments based on string patterns

In this section, we present two type of strings that may be subject to some efficient processing when hardcoded implementation is chosen as the basis for implementing

FAs. In [Kwk03a], we showed that for a single state of some randomly generated automaton, hardcoding using jump table outperformed the traditional table-driven implementation. In the subsection below, we extend the experiment to an accepting string that simply remains on a single state during the entire recognition process. The second case considers the conditions under which hardcode will continue to outperform the table driven version, even if the strings become relatively long. This is inferred from the results obtained from the next subsection and from the results relating to the threshold region (up to which hardcoded outperforms table-driven processing).

3.1 Strings that keep the FA in a single state

Let consider the automaton modelled in Figure 4 having 5 states with two accepting states 3 and 4. The strings *abab* and *cdef* of size 4 are both part of the language modelled by the automaton. In theory, the total time required to accept or reject each string should be roughly the same. However, we notice that for the string *abab*, once the device reads the first symbol *a*, it jumps to the final state 3 and remains there until the entire string is processed. On the other hand, the recognizer will transverse several different states in order to accept the string *cdef*. This observation indicates that in practice, the time required to accept strings of same size with different patterns may differ considerably from one another. The experiment randomly generated various

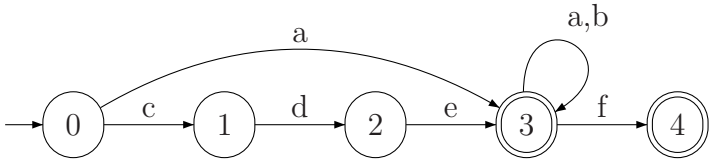

Figure 4: A state diagram that accepts the strings *abab* and *cdef*

automata of sizes between 10 and 1000 states using respectively 10, 15, 20, 25, 30, 35, 40, 45, and 50 symbols alphabet. Figure 5 depicts the difference in time between the table-driven and hardcoded implementation. It clearly shows that hardcode is consistently about 8 ccs faster than the table-driven implementation.

These results illustrate the fact that for strings following the pattern such as the one for *abab* above, the processor has sufficient space in its cache to hold the code relating to a single state. Since it always visits the same state over and over, no cache misses occur (neither for the hardcoded, nor the table-driven version) and there is a consequent high processing speed. It should be specifically noted that there are also no *data cache* misses in the the table-driven implementation. The scenario therefore represents the very best case behaviour for both the table driven and the hardcode implementations. It demonstrates that hardcode outperforms table-driven in this best case context.

The next experiment confirms that when long term behaviour tends towards best case behaviour, then the benefits of the hardcoded approach become increasingly apparent. This case is depicted in the next subsection.

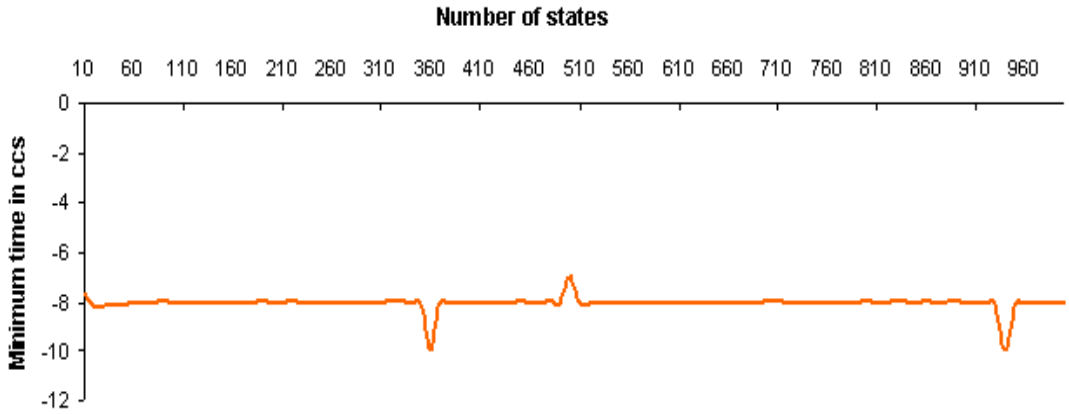

Figure 5: Comparative performance based on recurrent access on a single state

3.2 Strings that drive the FA randomly through a limited number of states

Consider the automaton modelled in figure 6. State t depicts some "sink state" in which the FA will remain after some other arbitrary set of states within the automaton have been visited. The state t is also assumed to be an accepting state. Based on the previous experiment, we would expect that the longer the FA remained in state t, the more the hardcoded implementation would enjoy an advantage over the table-driven version.

To verify this observation, and as a sort of sanity check on our results up to this point, hardcoded and table-driven implementations were set up to test strings of length $n-1$ in FAs with n states. The experiment was designed so that the behaviour in processing the first 300 states was random – in the same sense as previously described. However, thereafter the FA remains in the same state – i.e. the best case scenario prevails.

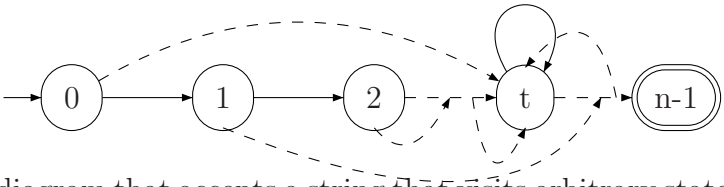

Figure 6: A state diagram that accepts a string that visits arbitrary states and remains on state t for some time

Figure 7 depicts the graphs obtained from the experiment. Unsurprisingly, it shows that hardcoding generally outperforms the table-driven implementation. However, there is also a suggestion in the data that in the longer term, the asymptotic improvement tends towards the 8ccs improvement observed in figure 5.

Of course, many more experiments similar to those described above could be run. An overall and general observation in regard to all these experiments is that they enable us to identify various ways in which the hardcoded implementation of

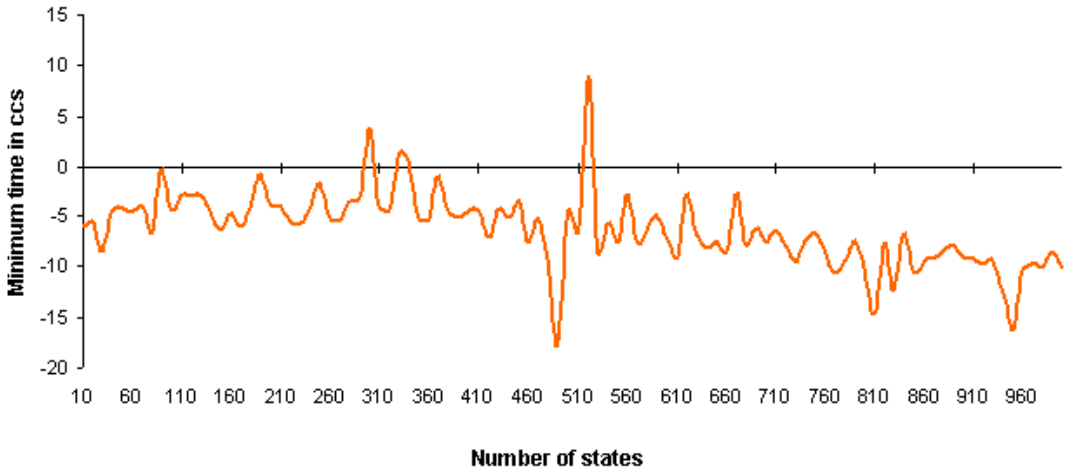

Figure 7: Comparison based on limited access on states

FAs may outperform the traditional table-driven implementation. The next section considers how such information could be used to capitalize on the advantages offered by both approaches.

4 A Preliminary Dynamic Algorithm

The experiments depicted in the previous sections clearly show that the efficiency of a string recognizer is highly dependent on the nature of the string being recognized. This suggests that if likely patterns of strings to be input are known in advance – at least in some probabilistic sense – then it may be possible to put in place a time optimizing mechanism to carry out the string recognition. Consequently, the idea of dynamically adapting the implementation strategy of the FA according to the expected input (or partially inspected) string may be considered. We use the acronym DIFAP to refer to this notion, designating Dynamic Implementation of FAs for Performance enhancement. Figure 8 depicts the overall design of a DIFAP system. When it is first invoked, the implementer provides the specification of the automaton to be used, regardless the type of string to be recognized. DIFAP then analyzes the specification and choose the appropriate way of implementing the automaton depending of the size of the derived automaton. In terms of the currently available data, if the size is less than 360 states, this means that hardcoding is likely to be the optimal approach in representing the automaton irrespective of the kind of string to be tested. On the other hand, if the size of the automaton is above 360 states, as suggested in Section 2, a hardcode implementation might be indicated if long term behaviour is likely to tend towards best case behaviour, a table-driven implementation will be the appropriate choice in the absence of such information. However, in the latter case, DIFAP relies on the kind of string received as input to adapt itself progressively to an implementation approach that is optimal in some sense (e.g. optimal in relation to the history of strings processed to date), resulting in improved average processing speed.

The figure indicates that for bigger automata size, a knowledge table (KT) is first checked. At this stage, we do not prescribe what information should be kept in the KT. We merely observe that the current input string could, in principle, undergo some preliminary scan to identify whether its overall structure conforms to some set of general patterns that favour hardcode over table-driven. If that is not the case, the table-driven version of the automaton specification is generated and is used by the recognizer to check whether the string is part of the language described by the FA or not. Otherwise, the hardcoded version of the FA's specification is generated and used for recognition.

Not indicated in the figure is the possibility of post-processing: after a string has been tested, the string and the test outcome could be used to update information in the KT. As a very simple example, we might decide to concretely implement the KT as a table of the FA's states, in which a count is kept of the number of times a state has been visited. This information could be used to rearrange the order of rows (which represent states) in the transition matrix used by the table-driven approach, in the hope of minimizing data cache misses when this implementation strategy is used. Alternatively, the same information could be used to dictate the blocks of hardcode that should preferentially be loaded into cache, in circumstances in which hardcoding is indicated. However, the foregoing should not be construed as the only way in which the KT can be implemented. We conjecture that there are many creative possibilities within this broad model that merit deeper investigation in the future.

One of the advantage of using such a dynamic algorithm is that the structure of the automaton does not always remains in the system after processing. Each automaton is always regenerated into its executable when the system is invoked. The only structure that permanently remains in the system is the algebraic specification of the automaton. This results therefore in some degree of minimization of memory load for automata of considerable size. However, there is no need to always regenerate automata of size less than 360 states since they will always be implemented in hardcode. That is the reason why in the figure no deletion of the generated hardcode is indicated when the "size less than 360" path is followed.

In an implementation of DIFAP, attention should be given to the following parts of the algorithm to minimize latencies:

- *Time to generate the recognizer:* Unless directly implemented by hand, any FA-related problem always requires a formal specification of the grammar that describes the automaton before its corresponding automaton is encoded. This is a general problem, and one specific to DIFAP. The DIFAP implementation could therefore use generator techniques similar to those used in efficient code generator tools such as YACC[2] which as been proven to be amongst the best tool available to create directly executable parsers. Unlike parsers, DIFAP's code generator will generate directly executable string recognizers.

- *Time taken to check the knowledge table:* One should take care to ensure that the matter of checking the KT does not degenerate into a time-inefficient exercise that negates any benefit from using the optimal string recognition strategy. Efficient algorithms should be devised that take minimal time to access the table and to chose the appropriate path to follow. This part of DIFAP may

[2]Yet Another Compilers Compiler

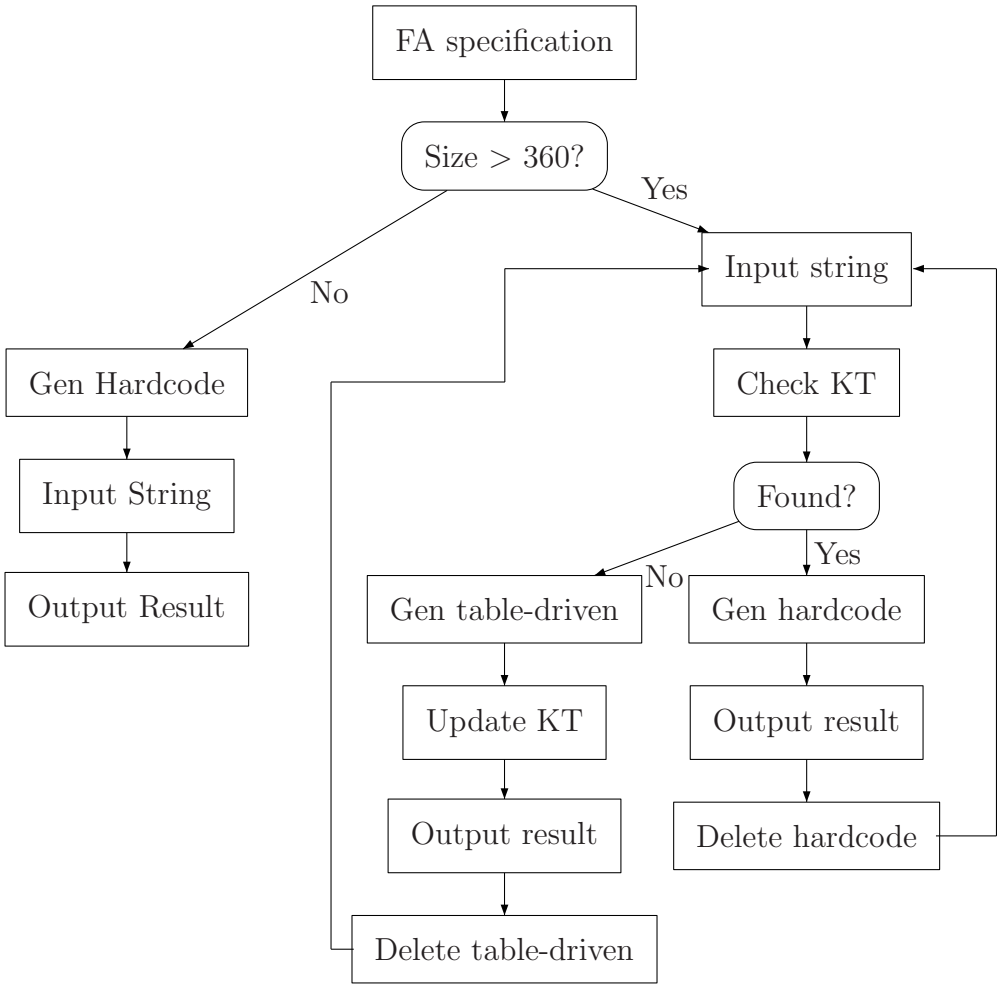

Figure 8: A Preliminary design of DIFAP

constitute a bottleneck. Intensive investigations will be made to provide an efficient approach to access the table and retrieve appropriate information.

- *Time required to update the knowledge table:* Although the precise nature and scope of the KT have not been identified here, it is envisaged that it will, itself, be a dynamic structure, changing over time in relation to the history of strings analyzed to date. However, there does not appear to be any reason for adapting the KT prior to processing the input string. Its update is something that can happen at a post-processing stage, and does not appear to be time-critical.

4.1 Applications of DIFAP

Finite automata are used to model several computational problems. Many of those problems are currently solved using the traditional table-driven approach. However,

it might be of great advantage to use the DIFAP model in order to overcome some speed latencies. The following is a high level survey of various kinds of problem that could benefit from the DIFAP approach:

- *Compiler construction:* Lexical analysis is the part of the compiler that deals with FAs. A lexical analyzer generator such as LEX usually generates table-driven code and uses it to scan the current identifier being converted into token. However, the automaton generated is often of relatively small size. This means that, instead of always using table-driven implementation, the compiler implementor could opt for the hardcoding implementation that might yield faster processing.

- *String Pattern Matching:* Even though a lot of work as been done to solve problems of exact and approximate string keyword pattern matching using efficient algorithms (e.g. [Wat95]) their practical implementation is sometimes very inefficient according to [Nr02]. In some circumstances, it might be beneficial to solve the problem of online string matching using hardcoding. Since many problems deal with a large amount of text, constructing a single automaton based on the text might be highly inefficient. Nonetheless, the text can be decomposed into small block on which we can derive its corresponding hardcode in order to achieve better processing.

- *Cellular automata* Implementing cellular automata to solve computational genetic problems requires, in general, small blocks of automata. Therefore, hardcoding might be of value in improving the processing speed of some of these problem where speed is a major factor.

- *Network Intrusion Detection:* In general, the size of the automaton generated depends on the variety of traffic to which the network is subject. Since Network Intrusion Detection algorithms can also construct FAs on the fly, DIFAP appears to be a suitable approach to adopt in such problems: the generated automaton will then be the one deemed to be most efficient in the given context.

- *Other FA application:* There are many other computational problems that rely on FA processing, where DIFAP could be of potential value. Aspects of natural language processing come time mind. However, it is not our intention to exhaustively enumerate all potential domains of application.

The foregoing indicates that there are indeed a significant number of problem domains that make the further elaboration of the DIFAP model a worthwhile endeavour. It is known that in many of these domains, efficient solutions have already been established. However, since DIFAP is a dynamic framework, we aim at improving the existing results using the technique. A investigation into each of the domain is therefore of concern but is out of the scope of this introductory work.

5 Conclusion and Future Work

In this paper, we have shown that the processing speed of a string recognizer not only depends on the length of the string being recognized but also on the kind of the

string under consideration. Alternative ways of implementing finite automata without relying on the traditional table-driven approach has been revisited after some intensive work done by Ketcha et al in [Kwk03a, Kwk03b, Ket03]. Ketcha's work showed that hardcoding of FAs might be a better ways of implementing FAs up to some threshold – approximately 360 states, in the context of the hardware and alphabet size used in those experiments. The various investigations performed has suggested the notion of Dynamic Implementation of Finite Automata for Performance. The DIFAP concept, provides automata implementers with an algorithm that is flexible enough to take advantage of both hardware and software considerations of the environment in which the string recognizer is being processed. The actual design of DIFAP relies on the constraints related to the automaton's structure as well as the kinds of string being processed. In the near future an implementation of DIFAP is envisaged, and various experiments will be conducted in order to characterize its efficiency. Other DIFAP issues not fully elaborated above will be to dynamically decide on matters such as the use of optimal data structures for particular kinds of FAs (e.g. linked list would appear to be a better way of representative very sparse transition functions, rather than a conventional two-dimensional matrix). Other DIFAP considerations include dynamic decisions about when to use stretched or jammed automata [Nwk03], and whether hybrid implementations that combine aspects of hardcode into a table driven implementation (or vice-versa) would be beneficial. However, these are all currently regarded as matters for future research. The final goal of DIFAP will be to design domain specific algorithms so that each problem can be solved on a highly efficient way.

References

[Kim02] Paul Kimmel. The Visual Basic .Net Developper's Book. Addison-Wesley, 2003.

[Ket03] E. Ketcha Ngassam. Hardcoding Finite Automata. MSC Dissertation. University of Pretoria, 2003.

[Kmp77] D. E Knuth and J.H Morris, Jr and V. R. Pratt. Fast Pattern Matching in Strings. SIAM J. Comput. Volume 6, 323-350, 1977.

[Kwk03a] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, Preliminary Experiments in Hardcoding Finite Automata, Poster paper, CIAA, Santa Barbara, 299-300, September 2003.

[Kwk03b] E. Ketcha Ngassam, Bruce. W. Watson, and Derrick. G. Kourie, Hardcoding Finite State Automata Processing, SAICSIT, Johannesburg, 111-121, September 2003.

[Nwk03] Noud De Beijer, Bruce W. Watson and Derrick G. Kourie, Stretching and Jamming of Automata, SAICIST, Johannesburg, 198-207, September 2003.

[Nr02] Gonzalo Navarro, and Mathieu Raffinot. Flexible Pattern Matching In String: Practical on-line search for texts and biological sequences. Cambridge University Press 2002.

[Wat95] Bruce W. Watson. Taxonomies and Toolkits of Regular Languages Algorithms. PhD Thesis. Technical University of Eindhoven, 1995.